Diplôme d'habilitation à diriger des recherches en sciences

Gilles Bernot

LIENS, CNRS URA 1327
45 rue d'Ulm
75230 PARIS Cedex 05 FRANCE.

(bernot@dmi.ens.fr ou bernot@frulm63.bitnet)

12 Février 1992

Remerciements

Je suis vivement reconnaissant à Egidio Astesiano d'avoir consacré une partie de son temps à lire ce document, mais aussi et surtout pour l'intérêt qu'il a accordé a mes travaux et les idées ou remarques dont il m'a fait profiter lors de nos rencontres.

Les travaux de Don Sannella ont toujours fait partie des « documents de référence » indispensables pour aborder les sujets qui me passionnent. Je le remercie chaleureusement de la sympathie qu'il a manifestée envers mon travail et de l'intérêt qu'il lui porte.

J'apprécie beaucoup l'approche de Pierre Lescanne vis à vis de la recherche. Ses critiques, remarques et conseils me sont précieux. Je l'en remercie vivement. L'intérêt qu'il porte à mes travaux est pour moi un honneur et me fait très plaisir.

Je tiens particulièrement à remercier Egidio Astesiano, Pierre Lescanne et Don Sannella d'avoir accepté de rapporter sur mon diplôme d'habilitation dans des délais très courts, malgré leurs nombreuses occupations. Que tous trois soient assurés de mon estime et de ma profonde reconnaissance.

Un point majeur dans la recherche est bien sûr avant tout le choix des sujets abordés. J'ai toujours trouvé au contact de Marie Claude Gaudel une mine inépuisable d'idées nouvelles. Ses conseils et son approche rationnelle m'ont souvent évité de tomber dans une attitude de pur théoricien loin des réalités informatiques. J'ai grand plaisir à travailler avec elle au LRI, et je lui accorde une confiance immense.

Depuis ma thèse de troisième cycle, j'ai toujours autant de plaisir à élaborer des théorèmes sur un coin de tableau avec Michel Bidoit. Ce sont pour moi des moments de choix que je trouve malheureusement beaucoup trop rares. Qu'il soit assuré de mon amitié.

Jean-Pierre Jouannaud m'a souvent fait profiter de ses conseils, avec une compétence que j'admire. Je le remercie vivement de la bienveillance qu'il m'accorde et d'avoir accepté de siéger dans ce jury.

Je suis particulièrement heureux de la participation de Guy Cousineau à ce jury. Il a toujours apporté un soutien amical à mon travail. Son excellent accueil lors de mon arrivée à l'ENS, puis ses conseils et ses encouragements, m'ont toujours été d'un grand secours.

Lors de nos rencontres, en France ou ailleurs, Pippo Scollo n'a jamais manqué de me faire profiter de ses idées ou de ses remarques. Je le remercie amicalement pour sa participation à ce jury et pour l'intérêt qu'il manifeste envers mes travaux.

Pascale Le Gall, Bruno Marre et Teodor Knapik, ainsi que Anne Deo-Blanchard et Pierre Dauchy, ont participé à divers titres aux travaux exposés ici. Lors de nos nombreuses discussions, ils ont énormément contribué à infléchir mes choix de recherche, à énoncer plus clairement les idées sous-jacentes, et à approfondir certaines théories. Je suis très heureux de la confiance qu'ils m'accordent et je les remercie de la franchise et de la clarté avec laquelle ils m'exposent leurs arguments.

Je remercie aussi tous les membres du LIENS et du LRI. Ils contribuent à faire de ces laboratoires des cadres de recherche exceptionnels.

Une part importante de mon travail est consacrée à l'enseignement. La qualité des cours que l'on dispense à l'ENS rend cette tâche immensément agréable. Cet environnement exceptionnel doit beaucoup à Michel Broué, Claude Puech et Guy Cousineau. Grâce à eux, on peut clore une année universitaire avec « la satisfaction du travail bien fait ». Je les en remercie tout particulièrement.

Je voudrais enfin remercier tous mes proches, excellents amis ou membres de ma famille, pour leur patience et leur soutien.

Table des chapitres

Chapitre 1 : Vue d'ensemble 1

Chapitre 2 : Correctness proofs for abstract implementations 45

Chapitre 3 : Testing against formal specifications : a theoretical view 75

Chapitre 4 : Software testing based on formal specifications :
a theory and a tool 95

Chapitre 5 : Proving the correctness of algebraically specified software :
modularity and observability issues 141

Chapitre 6 : Label algebras and exception handling 169

Chapitre 7 : Good functors ... are those preserving philosophy ! 227

(On trouvera une table des matières plus détaillée au début de chaque chapitre)

Chapitre 1 :
Vue d'ensemble

Table des matières

1 Introduction **2**

1.1 Le choix des spécifications algébriques 2

1.2 Les étapes de réalisation et de vérification 4

1.3 Les étapes de spécification et de validation 6

2 Implémentation abstraite **9**

3 Test de logiciel **13**

4 Sémantiques de l'observabilité **21**

5 Les algèbres étiquetées **29**

6 Les « métathéories » **34**

6.1 Sémantiques initiales et sémantiques « loose » 35

6.2 Sémantiques « loose » avec contraintes 37

6.3 Le paradigme « Institution Independent » 40

7 Conclusion **41**

1 Introduction

Le but de cette « Vue d'ensemble » est de décrire les idées majeures sous-jacentes aux travaux inclus dans ce document et de situer le cadre général dans lequel ils s'inscrivent. Nous tenterons ici d'expliquer aussi simplement que possible les motivations générales qui ont guidé les articles reproduits dans ce document (chapitres 2 à 7). Nous ne chercherons donc pas à énoncer dans ce chapitre 1 toutes les définitions, résultats ou preuves relatifs aux sujets abordés. Au contraire, nous ne donnerons l'énoncé d'une définition, ou d'un théorème, que s'il facilite la compréhension d'un point technique majeur. Les détails techniques pourront être retrouvés en lisant les articles fournis dans les chapitres suivants, pour lesquels cette vue d'ensemble peut être utilisée comme guide de lecture. Par contre, on trouvera quelquefois des remarques, contre-exemples ou problèmes ouverts qui ne figurent pas dans les articles ci-inclus. En particulier les sections 6.2 et 6.3 sont essentiellement dévolues à des problèmes ouverts qui devraient faire l'objet de recherches futures.

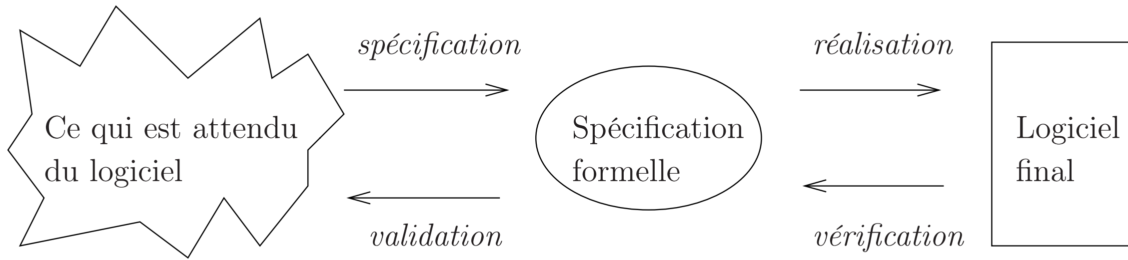
1.1 Le choix des spécifications algébriques

> Aphorisme : *Parler de la correction d'un logiciel sans l'avoir spécifié formellement revient à aborder la démonstration d'un théorème sans l'avoir énoncé clairement.*

Un enjeu fondamental du génie logiciel est la réalisation de logiciels « corrects », ou tout au moins exempts d'erreurs pouvant avoir des conséquences graves. Si l'on veut aborder cette question d'une manière rigoureuse, il faut bien évidemment fournir une réponse formelle à la question « qu'est-ce qu'un logiciel correct ? ». Elle est un préalable incontournable pour répondre à la question « comment assurer la correction d'un logiciel ? » (éventuellement partiellement, à défaut de mieux). Une réponse de bon sens à cette question préalable est « un logiciel correct est un logiciel qui fait ce que l'on attend de lui ». Il est donc avant tout nécessaire de *spécifier* ce que l'on attend d'un logiciel. On pourra alors seulement tenter de définir la correction d'un logiciel *par rapport à sa spécification.*

A l'heure actuelle, en pratique, les spécifications sont généralement fournies en langage naturel. Notre approche étant résolument formelle, le langage naturel ne peut être retenu car il est sujet à des ambiguïtés. Ces ambiguïtés ne sont pas nécessairement rédhibitoires pour le développement de logiciels de bonne qualité, mais, si nous voulons aborder rigoureusement cette question, il est nécessaire de considérer des spécifications dont le texte (i.e. la *syntaxe*) possède un sens précis (une *sémantique* mathématique). De telles spécifications sont appelées des *spécifications formelles*. Outre le fait que les spécifications formelles permettent une étude théorique de la notion de correction de logiciel, elles sont aussi, en pratique, un outil très utile pour réduire l'effort de développement de logiciel. En effet, passer d'une spécification informelle à une spécification formelle oblige du même coup à lever la plupart des ambiguïtés. On évite ainsi des interprétations erronées, ou simplement divergentes, des diverses équipes de développement ; ou au pire on les fait apparaître beaucoup plus tôt. On sait les coûts importants engendrés par des spécifications mal établies : une erreur découverte seulement lors de l'étape de codage coûte environ dix fois plus cher que lorsqu'elle est découverte lors de l'étape de spécification. Ceci conduit parfois (plus souvent qu'on ne le croit) jusqu'à l'abandon du projet. Par ailleurs (et nos efforts de recherche vont en ce sens), des spécifications formelles avec une syntaxe bien établie peuvent être manipulées via des techniques au moins partiellement automatisables.

Nous suivrons la terminologie suivante :



Notons que la partie informe de ce dessin (à gauche) n'a pas d'existence mathématique. C'est l'étape de *spécification* (la flèche en haut à gauche) qui a pour but de transformer, parfois partiellement, la description informelle de ce que l'on attend du logiciel en une spécification formelle (au milieu) qui, elle, possède une sémantique bien établie. L'étape de *réalisation* du logiciel, ou « implémentation »[1] (la flèche en haut à droite), est généralement une composition d'affinements successifs de la première spécification obtenue (dite « abstraite ») en des spécifications (de plus en plus « concrètes ») décrivant de plus en plus précisément les choix de mise en œuvre du logiciel (boîte de droite). Les véritables difficultés conceptuelles sont liées aux deux flèches en sens inverse (*vérification* et *validation*[2]). La *vérification* consiste à vérifier si le logiciel répond à sa spécification formelle. Malheureusement, un logiciel est un objet qui n'est pas rigoureusement modélisable de manière entièrement fidèle. En effet, dans le cadre du génie logiciel il faut le considérer au sein de son environnement (au sens large, en incluant aussi les aspects matériels par exemple). Il est donc nécessaire de faire bon nombre d'hypothèses pour pouvoir lui faire correspondre un objet mathématique (par exemple supposer la correction du système matériel qui le supporte, du compilateur, etc.). Ainsi la vérification ne peut être à proprement parler qu'un processus partiel. La situation est pire en ce qui concerne la *validation* (qui consiste à vérifier que la spécification formelle traduit correctement ce que l'on attend du logiciel). Puisque ce que l'on attend du logiciel n'est pas, par définition, exprimé formellement, la validation elle même n'est pas un processus formalisable.

Quel est dans ces conditions l'apport des spécifications formelles ? Au cours de l'étape de vérification, il est *justement* d'obliger à formuler les hypothèses nécessaires, et aussi d'en réduire le nombre. Nous verrons à quel point ceci est crucial pour le test de logiciel. De plus, une fois admise la sémantique d'un programme nous pouvons parfaitement caractériser ce que signifie sa correction par rapport à une spécification formelle. Au cours de l'étape de validation, l'apport d'une spécification formelle par rapport à une spécification en langage naturel réside en la connaissance assez complète de ses propriétés. Il est en particulier souvent possible d'extraire rapidement un prototype de la spécification formelle. Ce dernier permet de remettre éventuellement certains choix en cause avant que le logiciel ne soit développé. Il est également possible de prouver des propriétés additionnelles (théorèmes de la spécification formelle considérée). La présence, ou l'absence, de certaines propriétés peut remettre la spécification en cause avant de commencer le développement du logiciel.

Notons enfin qu'il n'existe pas d'approche universelle satisfaisante permettant de spécifier formellement *toutes* les propriétés pertinentes que l'on peut attendre d'un logiciel *quelconque*. On peut douter qu'une telle approche existe à moyen terme. En fait un « langage »formel universel existe déjà, c'est la mathématique elle même. Disons pour être plus précis la théorie des ensembles (c'est presque le choix fait pour le langage Z [MN90]), ou celle des catégories [McL71]. Le langage mathématique pur est malheureusement beaucoup trop riche[3] pour être utilisable car

[1] Nous devrions dire « implantation », mais l'usage actuel tend à suivre l'anglais.

[2] La terminologie que nous suivons n'est pas universellement établie ; certains auteurs inversent même les rôles respectifs de « vérification » et « validation ».

[3] tout comme le langage naturel finalement. . .

une bonne méthode de spécification doit autoriser des techniques simples de preuve, d'aide à la spécification, etc. On est donc conduit à considérer des approches plus spécialisées, qui sont plus restrictives mais offrent du même coup une meilleure aide à la spécification et de meilleurs outils de vérification ou de validation. Selon l'application envisagée, il est plus judicieux de choisir une approche formelle plutôt qu'une autre, voire d'en considérer plusieurs afin de traiter divers aspects complémentaires. Parmi toutes les approches possibles des spécifications formelles, citons par exemple les réseaux de Pétri, les automates, les systèmes de transitions, diverses logiques (premier ordre, ordre supérieur, temporelle...), la théorie des graphes, etc.

Nous nous focaliserons ici essentiellement sur les *types abstraits algébriques* (i.e. sur les théories de *spécifications algébriques*). Cette approche est particulièrement efficace pour décrire les fonctionnalités attendues d'un logiciel. Elle conduit à un style de spécification relativement bien adapté aux logiciels de grande taille, en particulier parce que la structuration des spécifications peut être étudiée de manière assez simple, mais aussi parce qu'elle bénéficie d'un contexte fortement typé. Elle est par contre peu utilisable pour spécifier des applications où des aspects de « temps réel » interviennent. Il est assez difficile de traduire des propriétés mettant en jeu des mécanismes concurrents ou du parallélisme. Par conséquent, certaines applications (telles que les interfaces homme-machine) ne peuvent pas être entièrement spécifiées algébriquement. On voit donc que ce choix des spécifications algébriques est à la fois partial (d'autres approches des spécifications formelles sont tout aussi intéressantes) et partiel (on ne peut pas tout spécifier avec les types abstraits algébriques). Il est toutefois bien clair qu'il faut focaliser assez nettement les théories de spécification étudiées afin d'obtenir des résultats applicables.

1.2 Les étapes de réalisation et de vérification

Il s'agit ici de réaliser un logiciel à partir de sa spécification algébrique et de s'assurer (au moins partiellement) de la correction du logiciel obtenu, par rapport à cette spécification. On suppose donc que « ce que l'on attend du logiciel » a déjà été, au moins partiellement, transcrit sous forme d'une spécification algébrique. Remarquons que la notion de correction du logiciel est alors fortement dépendante de la sémantique choisie (sémantique initiale, « loose », observationnelle ou autre, comme on le verra plus loin).

Il existe trois classes d'approches :

- celles qui tentent d'assurer la correction du logiciel *a priori*. Elles sont fondées sur des techniques d'extraction de programmes à partir de spécifications (plus ou moins automatisées selon les cas).
- celles qui tendent à vérifier, éventuellement partiellement, la correction d'un logiciel *a posteriori*. Elles peuvent faire appel à des techniques de preuve ou de test.
- enfin celles fondées sur la *réification* de spécifications. Elles consistent à établir des spécifications de plus en plus précises, à partir de la spécification abstraite de départ. Les spécifications intermédiaires traduisent de plus en plus de choix concrets de mise en œuvre (incluant des choix de découpage en modules). On itère le processus jusqu'à arriver à une spécification « concrète », pour laquelle il est relativement aisé de construire un logiciel qui la réalise.

La première approche peut être fondée sur des techniques de *décomposition* issues de la structure des types abstraits de données mis en jeu dans la spécification. Ces techniques permettent d'engendrer des squelettes de programme via des schémas de décomposition, c'est par exemple le cas de Caty [Gre84] au sein de l'environnement de spécification Asspegique [BCGKS87]. D'autres méthodes plus complètes sont fondées sur l'expression d'un principe d'induction pour chaque type abstrait de données (souvent exprimable automatiquement à partir de la connais-

sance d'un ensemble de *constructeurs libres* de cette structure). Elles utilisent alors des techniques de résolution et font appel à des extensions de PROLOG [Fri90].
D'autres enfin utilisent des logiques constructives intuitionnistes d'ordre supérieur. Elles utilisent aussi des principes d'induction pour chaque type abstrait de données mis en jeu dans la spécification. La méthode est alors de « prouver la spécification », plus précisément de prouver la « réalisabilité » de la spécification[4], et d'extraire de cette preuve un programme réalisant la spécification (puisqu'il s'agit d'une logique constructive). C'est le cas de la théorie des constructions [CH85]. La difficulté majeure de cette approche est d'extraire d'une preuve les étapes possédant un réel contenu algorithmique.
A l'heure actuelle, les méthodes de construction de programmes corrects *a priori* sont peu utilisées dans le cadre du génie logiciel, en particulier parce que les programmes résultants sont assez peu performants[5]. C'est par contre très certainement une approche prometteuse à court terme dans le cadre du prototypage de spécifications. Malgré son intérêt évident, nous n'aborderons pas plus avant cette approche dans ce document.

La seconde approche (vérification a posteriori) peut faire appel à des techniques de preuve. On utilise alors des logiques de Hoare pour les langages impératifs, ou simplement la définition de la sémantique du langage pour les langages fonctionnels. Pour les langages logiques, le style déclaratif commun aux spécifications algébriques et à un langage tel que PROLOG facilite les preuves ; il s'agit essentiellement d'établir une correspondance entre leurs sémantiques, qui sont très proches (ceci a été en particulier étudié pour des langages tels que SLOG [Fri85] ou RAP [GH86]). La vérification a posteriori peut aussi faire appel à des techniques de test. Il s'agit alors de vérification partielle. Les techniques de test sont cependant les plus utilisées en pratique. Lorsque le test est étudié sur des bases rigoureuses, le choix des jeux de tests est souvent fondé sur la structure interne du logiciel sous test (test dit « boîte translucide » ou « boîte blanche »). Par contre, le test dit « boîte noire », pour lequel la sélection des jeux de tests repose sur la spécification, était jusqu'à très récemment peu formalisé. Ceci était essentiellement une conséquence de l'usage de spécifications informelles. Nous verrons plus loin l'apport considérable des spécifications formelles dans ce domaine.

Il est important de noter que le test est la seule méthode de vérification qui permette de traiter la correction d'un logiciel dans son ensemble, c'est-à-dire intégré dans son environnement (au sens large, cf. la discussion de la section 1.1, page 3). Il est donc indispensable, dans le cadre du génie logiciel, d'associer test et preuve dans le processus de vérification. Les techniques de preuve à elles seules, bien que très fiables, ne peuvent en aucun cas être suffisantes (au moins à l'heure actuelle) puisqu'elles ne prennent pas en compte l'environnement complet.

La troisième approche enfin (réification) est souvent utilisée en pratique, même pour des spécifications informelles. C'est entre autres le seul moyen de découper judicieusement un logiciel en modules de taille raisonnable. Dans le cadre des spécifications algébriques qui nous intéresse ici, la réification de spécifications est appelée *implémentation abstraite* [EKMP82]. Etant donnée une spécification de départ, dite *abstraite*, on propose une spécification implémentant les fonctionnalités requises grâce à plusieurs structures de données de plus bas niveau. On doit alors prouver que la seconde spécification, dite *concrète*, « simule »correctement la spécification abstraite. On itère ce processus jusqu'à obtenir un découpage suffisamment fin, et des fonctions suffisamment simples, pour être soumis à l'étape de programmation. Il s'agit donc essentiellement d'une approche effectuant des vérifications a posteriori, mais les étapes de réification sont suffisamment élémentaires pour simplifier les preuves ou le test de chacune d'elles. Il existe cependant des méthodes, souvent fondées sur la forme des signatures et pouvant utiliser la théorie des graphes, pour

[4] C'est-à-dire l'existence d'un programme qui la satisfait, par exemple exprimé sous forme d'un λ-terme.
[5] On peut cependant améliorer leurs performances en élaborant des principes d'induction ad hoc.

guider les choix d'implémentation abstraite. Nous n'aborderons pas ce sujet ici[6]. Les remarques faites à propos de la vérification a posteriori s'appliquent donc aussi à cette troisième approche. En fait, aussi bien les techniques de preuve que celles de test présupposent généralement que la structure modulaire de la spécification algébrique est proche de celle du logiciel la réalisant (ou pour être plus précis du logiciel candidat à sa réalisation correcte...). Ceci permet de découper la vérification en plusieurs vérifications plus simples, module par module. On conçoit donc bien qu'il faut pour cela vérifier un logiciel par rapport à une spécification déjà « assez concrète », les spécifications plus abstraites pouvant être utilisées seulement lors de l'intégration des modules.

1.3 Les étapes de spécification et de validation

> Aphorisme : *Une spécification abstraite doit exprimer le « quoi », c'est-à-dire ce que l'on attend d'un logiciel, et non le « comment », c'est-à-dire les choix concrets de mise en œuvre (algorithmes, etc.).*

Il s'agit ici de faciliter l'écriture d'une spécification traduisant ce que l'on attend d'un logiciel et de fournir des sémantiques adaptées (c'est-à-dire aptes à caractériser aussi fidèlement que possible l'ensemble des réalisations correctes d'une spécification).

Rappelons que, contrairement à l'étape de vérification où des preuves de correction de logiciel sont envisageables (une fois admise la correction du compilateur, de l'environnement, etc.), des preuves de correction d'une spécification abstraite (en validation) n'ont aucun sens. On ne pourrait *prouver* la correction (la validité) d'une spécification abstraite que par rapport à une formalisation de « ce que l'on attend du logiciel », or par définition « ce que l'on attend du logiciel » (partie gauche du dessin page 3) n'est pas formellement défini lors des étapes de spécification et de validation. C'est justement le rôle de ces étapes que d'en fournir une description formelle, éventuellement partielle. Il en résulte que, pour justifier tel ou tel choix de sémantique ou d'écriture de spécification, nous ne pourrons donner que des arguments « de bon sens »mais non formels. Nous allons maintenant mentionner les arguments majeurs qui permettent de choisir une sémantique algébrique plutôt qu'une autre.

Avant tout, il est important que la sémantique soit proche de « l'intuition » fournie par les axiomes. Le point clef est donc la définition d'une correspondance bien choisie entre les axiomes écrits et les modèles validant ces axiomes. Cette correspondance doit permettre de spécifier les *propriétés abstraites attendues* et non pas un *moyen de les réaliser*. Pour donner un exemple simple, spécifier une opération de tri par un quelconque algorithme de tri, même le plus simple, n'est pas une spécification abstraite acceptable. Une telle spécification ne devrait être fournie qu'après au moins une étape de réification. Une bonne spécification abstraite d'un tri doit se contenter d'affirmer que le résultat est une permutation de l'entrée, et est trié. C'est seulement le rôle de l'étape de vérification que de prouver la correction d'un algorithme de tri particulier. De la même façon, lors du découpage d'une spécification en modules, il n'est pas judicieux de devoir écrire certains axiomes dans le seul but de préserver la consistance ou la complétude d'un module par rapport aux modules qu'il utilise. Cette propriété doit au contraire être un élément constructif pour faciliter la spécification. Prenons là encore une exemple simple : supposons que l'on écrive un module *Ensembles* important un module *Elements* et que l'on veuille spécifier une opération de choix d'un élément dans un ensemble (*choose*). La seule propriété pertinente que l'on puisse accepter de spécifier au niveau le plus abstrait est que $choose(E)$ appartient à E pour tout ensemble E non vide. C'est à la sémantique d'imposer implicitement que $choose(E)$ soit un

[6] Je ne donnerai même qu'une description très rapide de l'implémentation abstraite, mes premiers travaux à ce sujet ayant été décrits dans ma thèse de troisième cycle.

élément préexistant, et non pas une nouvelle valeur de type *Elements* engendrée par l'opération *choose* ; ceci bien que le choix exact de cet élément ne soit pas précisé par les axiomes. On conçoit donc bien que le choix d'une sémantique est crucial lors des étapes de spécification et de validation, puisque c'est seulement au travers de celle-ci qu'un axiome reflète (ou ne reflète pas) fidèlement ce que l'on attend d'un logiciel.

Un autre critère important est le pouvoir d'expression de la théorie de types abstraits algébriques choisie. Il est entre autres indispensable de pouvoir spécifier les cas exceptionnels ainsi que des traitements d'exceptions. Il faut par exemple pouvoir spécifier la levée d'un message d'erreur pour $choose(\emptyset)$. De même, la variété des connecteurs logiques acceptés dans la syntaxe d'une spécification est un élément important pour la concision des spécifications abstraites. Etre obligé d'écrire de nombreux axiomes pour contourner le manque de mécanismes de traitement d'exceptions, le manque de connecteurs, etc, nuit à la compréhension d'une spécification, et donc à sa validation.

Pour valider une spécification, un moyen efficace serait d'expérimenter une réalisation correcte de cette spécification... Bien sûr il n'est pas réaliste d'attendre l'étape de codage du logiciel pour remettre en cause tel ou tel choix de spécification. C'est pourquoi la possibilité de *prototyper* une spécification abstraite, même partiellement, est d'une utilité majeure. Ceci permet de modifier au plus tôt des choix mal adaptés. Malheureusement, il est bien évident que le prototypage est dans une large mesure antagoniste avec les deux critères mentionnés précédemment, en particulier celui selon lequel une spécification abstraite doit exprimer les propriétés attendues et non pas un quelconque moyen de réalisation du logiciel. Il n'en reste pas moins qu'un tel argument peut être acceptable pour restreindre légèrement le pouvoir d'expression dans certains cas. Notons toutefois que ceci n'est pas aussi restrictif qu'on pourrait le croire : un prototype n'est pas nécessairement extrait d'une spécification algébrique via des outils aussi élémentaires que l'algorithme de Knuth-Bendix... En fait, toutes les approches de « correction a priori » décrites dans la section 1.2 fournissent des techniques de prototypage très puissantes. De plus, rien n'impose qu'un prototype soit une réalisation complète d'une spécification. On peut parfaitement accepter des prototypes partiels, ils auront au moins l'avantage de « donner une idée » de la sémantique d'une spécification. De même que pour le test (voir plus loin) ce qui est important dans ce cas est assurément de bien préciser *ce que le prototype ne fait pas*[7]. Peut-être même pourrait-on engendrer des prototypes à partir d'une spécification et « d'hypothèses » similaires à celles utilisées pour le test, exprimant les fonctionnalités que le prototype ne remplit pas pleinement. Nous ne développerons pas cette question dans ce document.

Le prototypage n'est pas le seul moyen d'expérimenter une spécification abstraite. Lorsque l'on spécifie, il existe de nombreuses propriétés que l'on n'écrit pas mais qui, selon le « spécifieur », devraient être des conséquences de la spécification (absence de blocage, de situation catastrophique, etc). Un moyen d'expérimenter la spécification est alors tout simplement d'essayer de *prouver* ces propriétés additionnelles. C'est par exemple une approche utilisée dans LARCH [GHW85]. Il en résulte que l'existence de règles d'inférences consistantes par rapport à la sémantique choisie (à défaut de complètes) est un critère majeur pour faciliter l'étape de validation. Bien évidemment, sitôt que la sémantique choisie offre un pouvoir d'expression satisfaisant (relativement aux deux premiers critères cités) elle s'accompagne de nombreux cas d'indécidabilité. Toutefois, en pratique, on peut souvent remettre en question la spécification si la preuve d'une propriété, que l'on pense conséquence directe de la spécification, échoue.

Remarquons enfin que, bien que la théorie du test que nous développons dans ce document

[7]Bien qu'à ma connaissance aucune recherche n'ait été effectuée dans ce sens, je crois qu'une approche du prototypage s'inspirant des méthodes que nous utilisons pour le test serait prometteuse.

soit centrée sur le test de *logiciel*, les jeux de tests engendrés par notre méthode peuvent aussi être utilisés pour fournir des « exemples typiques » utiles aussi pour la validation de la *spécification* elle même.

Plan

Tous les travaux décrits dans ce document sont centrés autour des spécifications algébriques. La discussion générale informelle qui précède nous permet de les classer selon deux motivations majeures : ceux qui tendent à faciliter les étapes de réalisation et de vérification, et ceux qui tendent à faciliter les étapes de spécification et de validation.

La section 2 contient une description rapide des idées majeures ayant conduit à la théorie de l'implémentation abstraite développée dans l'article intitulé « *Correctness proofs for abstract implementation* »[8], et cet article constitue le chapitre 2 de ce document. La section 3 présente une synthèse de la théorie du test de logiciel fondé sur les spécifications algébriques développée dans « *Testing against formal specifications : a theoretical view* »[9] et de l'approche générale développée avec Bruno Marre et Marie-Claude Gaudel dans « *Software testing based on formal specifications : a theory and a tool* »[10]. Ces articles constituent respectivement les chapitres 3 et 4 de ce document. Les sections 2 et 3 présentent donc des approches complémentaires pour la réalisation et la vérification de logiciel. Elles permettent d'une part de concevoir un logiciel par étapes de réification successives dont la correction peut être démontrée (implémentation abstraite) et d'autre part de vérifier partiellement, en le testant, un logiciel (ou un module de programme) au sein de son environnement, par rapport à sa spécification (ou au module de spécification qui lui correspond).

Les sections 4 à 6 seront consacrées à l'étude de diverses sémantiques des spécifications algébriques. Comme nous l'avons déjà dit, notre objectif est de permettre d'écrire des spécifications aussi proches que possible de « l'intuition », afin de faciliter les étapes de spécification et de validation. La section 4 décrit brièvement une sémantique observationnelle et modulaire pour les spécifications algébriques. L'article correspondant intitulé « *Proving the correctness of algebraically specified software : modularity and observability issues* »[11], écrit avec Michel Bidoit, constitue le chapitre 5 de ce document. Nous décrivons ensuite une théorie assez générale des types abstraits algébriques qui permet, entre autres, de donner une sémantique puissante avec traitement d'exceptions. Elle a été développée en collaboration avec Pascale Le Gall, et l'article intitulé « *Label algebras and exception handling* »[12] est reproduit en chapitre 6. Enfin la section 6 contient une discussion sur le bien fondé de certaines « métathéories » souvent utilisées pour produire des résultats supposés indépendants de la théorie des types abstraits algébriques considérée. Nous motiverons tout d'abord, en section 6.1, le développement de sémantiques modulaires dites « loose » (résultats et contre-exemples établis dans l'article « *Good functors are those preserving philosophy* »[13], reproduit en chapitre 7), et nous verrons quelques arguments pour utiliser des contraintes sémantiques similaires à celles développées dans la « sémantique par classes stratifiées de modèles ». Les sections 6.2 et 6.3 poursuivent cette approche. La section 6.2 démontre que la « sémantique par classes stratifiées de modèles » telle qu'elle est définie actuelle-

[8]Paru dans le journal « Information and Computation », Vol.80, No.2, p.121-151, en Février 1989.

[9]Paru dans les Proc. TAPSOFT CCPSD, Brighton UK, Springer-Verlag LNCS 494, p.99-119, en Avril 1991.

[10]Paru dans « *Software Engineering Journal* », Vol.6, No.6, p.387-405, en Novembre 1991..

[11]Une version légèrement plus courte a paru dans les Proc. de la conf. AMAST-2, Iowa, en Mai 1991, à paraître dans LNCS.

[12]Version longue non encore soumise.

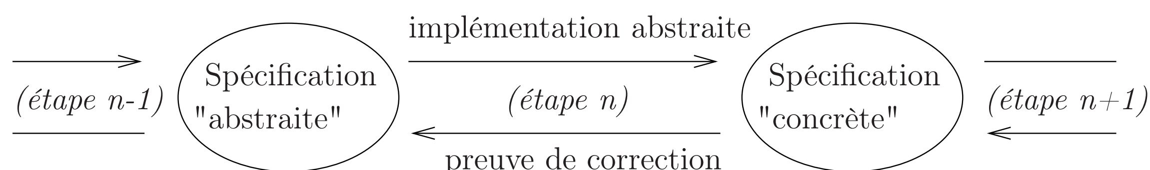
[13]Paru dans les Proc. de la « Summer conference on category theory and computer science », Springer-Verlag LNCS 283, p.182-195, en Septembre 1987.

ment n'est pas assez générale pour être instanciée par une sémantique *quelconque*, en particulier par les types abstraits algébriques avec traitement d'exceptions. La section 6.3 démontre entre autres que les « métathéories » existant actuellement ne sont pas suffisantes pour développer une théorie du test indépendante de la sémantique algébrique choisie. Ces deux questions ouvertes cloront notre « introspection dans le monde des types abstraits algébriques » et une conclusion brève sera fournie en section 7.

2 Implémentation abstraite

Les détails techniques relatifs à cette section sont consignés dans l'article intitulé « *Correctness proofs for abstract implementation* » reproduit en chapitre 2.

La réalisation et la vérification via des méthodes d'*implémentations abstraites* se situent dans le cadre d'un développement par « réifications » successives. Puisque nous sommes au sein des étapes de réalisation et de vérification, on suppose déjà disposer d'une première spécification algébrique (la plus « abstraite ») de ce que l'on attend du logiciel. Le but est alors de préciser pas à pas les choix de mise en œuvre du logiciel via plusieurs implémentations abstraites successives conduisant à des spécifications de plus en plus « concrètes ». L'idée est d'itérer ces étapes d'implémentation abstraite jusqu'à aboutir à des spécifications fondées seulement sur des structures de données relativement communes, ou fournies parmi les types de données standard du langage de programmation. En première approche, une étape élémentaire considère donc deux spécifications dont l'une sera dite « abstraite » et l'autre « concrète ». Un avantage majeur de cette méthode est que chaque implémentation abstraite ne met en jeu *que* des spécifications algébriques. Nous nous trouvons donc dans un cadre théorique homogène, au sein duquel on peut envisager relativement facilement de faire des preuves de correction.



Pour fixer les idées, nous allons décrire ci-dessous un exemple relativement simple. Il est néanmoins choisi pour soulever les trois problèmes typiques de l'implémentation abstraite. Nous adopterons de plus ici une sémantique initiale (c'est-à-dire que le modèle « naturel »auquel on se réfère est l'algèbre initiale).

Considérons tout d'abord la spécification abstraite des multi-ensembles de la figure 2. Nous supposons que la spécification des éléments (la sorte *Elem*) est fournie par ailleurs, ainsi que celle des booléens (la sorte *Bool*). Nous noterons ELEM et BOOL ces spécifications, que nous ne reproduirons donc pas ici. Nous supposons de plus que *Elem* est munie d'un prédicat d'égalité *eq*. La spécification MULTI est très classique ; notons simplement que l'opération *remove* n'enlève pas toutes les occurrences d'un élément, mais seulement une d'entre elles, si l'élément est présent. Pour éviter toute complication due au traitement d'exceptions, nous nous plaçons ici dans le cadre simple de la sémantique du groupe ADJ [GTW78] et nous spécifions que *remove* est réduite à l'identité si l'élément n'est pas dans l'ensemble.

Supposons maintenant que l'on choisisse d'implémenter abstraitement les multi-ensembles par des listes de couples formés d'un élément et d'un entier relatif (nous ne donnerons pas non plus la spécification RELAT des entiers relatifs. . .). Pour chaque couple (x, z) de la liste, z est alors le nombre d'occurrences de l'élément x dans le multi-ensemble représenté. Il est donc clair que z ne sera jamais négatif (ni même nul). Cette implémentation est cependant parfaitement

1— Spécification abstraite des multi-ensembles

Spec « MULTI ».

S : $\{Multi\}$

Σ : $\emptyset\ :\to Multi$

$insert\ :\ Elem \times Multi \to Multi$

$remove\ :\ Elem \times Multi \to Multi$

$\subseteq\ :\ Multi \times Multi \to Bool$

Ax :

$$insert(x, insert(y, M)) = insert(y, insert(x, M))$$
$$remove(x, \emptyset) = \emptyset$$
$$remove(x, insert(x, M)) = M$$
$$eq(x, y) = false \implies remove(x, insert(y, M)) = insert(y, remove(x, M))$$
$$\emptyset \subseteq M = true$$
$$insert(x, M) \subseteq \emptyset = false$$
$$M \subseteq insert(x, N) = remove(x, M) \subseteq N$$

réaliste (par exemple lorsque les entiers relatifs sont les seuls entiers fournis par le langage de programmation). Les structures de données utilisées pour l'implémentation abstraite peuvent être spécifiées abstraitement comme dans la figure 2.

2— Spécification abstraite des listes de couples

Spec « LISTE ».

S : $\{Couple\ ,\ Liste\}$

Σ : $(_,_)\ :\ Elem \times Relat \to Couple$

$[\,]\ :\to Liste$

$_ :: _\ :\ Couple \times Liste \to Liste$

Ax : vide

Un multi-ensemble étant alors représenté par une liste, il faut une « opération de coercion » transformant une liste en un ensemble. Dans le cadre de l'implémentation abstraite, cette opération est appelée l'*opération d'abstraction*, notée A. Remarquons que de manière générale, cette opération d'abstraction n'est pas nécessairement une coercion. Elle peut être un produit cartésien de plusieurs types de données « concrets » (c'est par exemple le cas lorsque l'on implémente des piles par des couples formés d'un tableau et d'un entier).

L'implémentation abstraite à proprement parler peut être spécifiée comme décrit en figure 2. Cette spécification utilise la spécification LISTE, et possède a priori la même signature que la spécification MULTI, enrichie de l'abstraction A. Nous pouvons maintenant préciser le schéma donné au début de cette section, en page 9. On remarque tout d'abord que les spécifications LISTE et MULTI partagent les spécifications ELEM et BOOL. On peut aussi noter que, alors que les spécifications MULTI, LISTE, ELEM, BOOL et RELAT sont de « vraies » spécifications *abstraites* qui ne décrivent aucun choix de mise en œuvre, la spécification IMPL ne répond pas à ce principe. IMPL décrit de réels choix d'implémentation, elle peut être en un certain sens qualifiée de *concrète*.

3— Implémentation abstraite de MULTI par LISTE

Spec « IMPL ».

$S:\quad \{Multi\}$

$\Sigma:\quad A\ :\ Liste \to Multi$

$\emptyset\ :\to Multi$

$insert\ :\ Elem \times Multi \to Multi$

$remove\ :\ Elem \times Multi \to Multi$

$\subseteq:\ Multi \times Multi \to Bool$

$Ax:\quad \emptyset = A([\,])$

$insert(x, A([\,])) = A((x,1) :: [\,])$

$insert(x, A((x,z) :: L)) = A((x,z+1) :: L)$

$eq(x,y) = false\ \wedge\ insert(x, A(L)) = A(L') \Longrightarrow$

$insert(x, A((y,z) :: L)) = A((y,z) :: L')$

$remove(x, A([\,])) = A([\,])$

$2 \leq z = true \Longrightarrow remove(x, A((x,z) :: L)) = A((x,z-1) :: L)$

$2 \leq z = false \Longrightarrow remove(x, A((x,z) :: L)) = A(L)$

$eq(x,y) = false\ \wedge\ remove(x, A(L)) = A(L') \Longrightarrow$

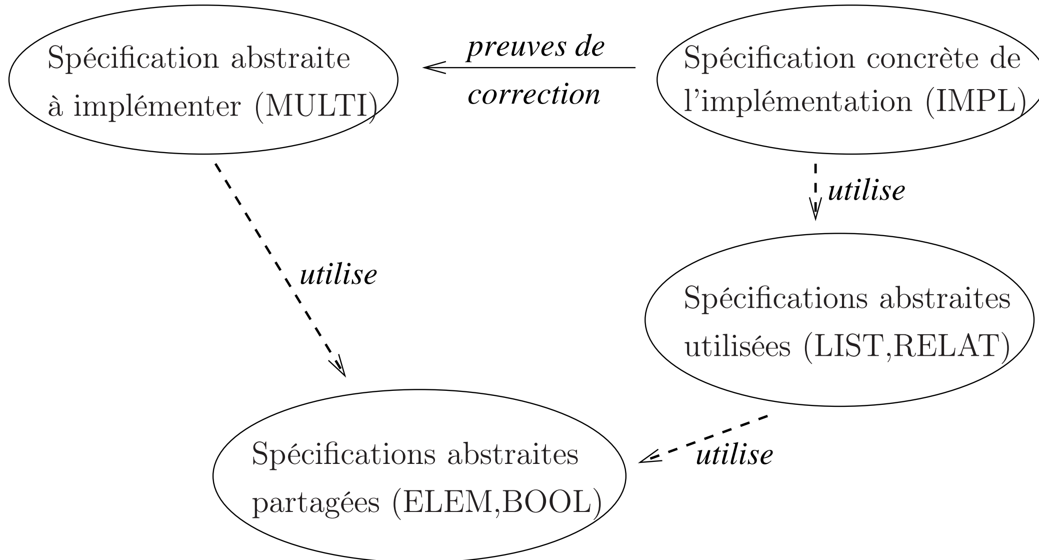
$remove(x, A((y,z) :: L)) = A((y,z) :: L')$

$A([\,]) \subseteq A(L) = true$

$A((x,z) :: L) \subseteq A([\,]) = false$

$2 \leq z = true \Longrightarrow A(L) \subseteq A((x,z) :: L') = remove(x, A(L)) \subseteq A((x,z-1) :: L')$

$2 \leq z = false \Longrightarrow A(L) \subseteq A((x,z) :: L') = remove(x, A(L)) \subseteq A(L')$



Il faut définir la correction d'une telle implémentation abstraite. Ceci soulève trois problèmes maintenant classiques. L'implémentation abstraite IMPL décrite en figure 2 doit clairement être considérée comme correcte, pourtant :

– La structure de données spécifiée par IMPL contient des éléments qui ne doivent clairement pas être considérés comme des multi-ensembles acceptables. Par exemple une liste de la forme $(x,-1) :: [\,]$ ne représente aucun multi-ensemble ; c'est aussi le cas de $(x,1) :: (x,1) :: [\,]$. Ceci signifie que les éléments $A((x,-1) :: [\,])$ et $A((x,1) :: (x,1) :: [\,])$, bien que de type $Multi$, devraient être ignorés. Notons que l'on peut toutefois aisément caractériser les « vrais » multi-ensembles : ce sont ceux qui sont atteints par des termes construits au-

dessus de la signature de MULTI (i.e. des termes ne contenant pas A). Les éléments non atteints par une certaine signature sont souvent appelés des « junks » (même en Français). On pourrait tenter de considérer l'abstraction A comme une fonction partielle, mais ceci conduit à des preuves de correction très complexes où le domaine de définition de A doit être explicitement caractérisé. Cette caractérisation s'appelle les *invariants de représentation*, et ceux-ci peuvent malheureusement mettre en jeu des formules trop complexes pour pouvoir être exprimées dans le cadre homogène des types abstraits algébriques sans ajouter des « opérations cachées ».

- La plupart des algèbres satisfaisant la spécification IMPL ne satisfont pas les axiomes de MULTI. Par exemple la permutabilité de *insert* : rien n'impose dans IMPL que $A((x,z) :: (y,z') :: L)$ soit égal à $A((y,z') :: (x,z) :: L)$. On voit donc que l'égalité des structures « concrètes » ne coïncide pas avec l'égalité abstraite, mais en est seulement un sous-ensemble. Contrairement aux invariants de représentation, il est possible dans la plupart des cas de caractériser de manière simple quand deux valeurs concrètes représentent la même valeur abstraite. Les axiomes correspondants sont appelés la *représentation de l'égalité*. L'article du chapitre 2 démontre qu'ils peuvent être utilisés efficacement pour prouver la correction d'une implémentation.
- Il est cependant impossible de quotienter « brutalement » les algèbres satisfaisant IMPL par la représentation de l'égalité afin de prouver que les axiomes de MULTI sont satisfaits. En effet, il faut d'abord supprimer tous les « junks » mentionnés dans le premier point. Sinon des inconsistances majeures peuvent apparaître. Raisonnons par l'absurde et supposons qu'après avoir quotienté par la représentation de l'égalité les axiomes de MULTI soient satisfaits. L'axiome de MULTI

$$insert(x,M) \subseteq \emptyset = false$$

pourrait alors être appliqué à l'élément non atteignable $M_0 = A((x,-1) :: [\,])$, et l'axiome

$$remove(x, insert(x,M)) = M$$

impliquerait en particulier que

$$insert(x,M_0) = remove(x, insert(x, insert(x,M_0)))$$

Or, via les axiomes de IMPL, on a :

$$remove(x, insert(x, insert(x,M_0))) = remove(x, A((x,1) :: [\,])) = A([\,]) = \emptyset$$

Il vient donc l'inconsistance suivante :

$$false \;=\; (insert(x,M_0) \subseteq \emptyset) \;=\; (\emptyset \subseteq \emptyset) \;=\; true$$

La question qui se pose est par conséquent *« comment éliminer les junks de manière simple ? »* en particulier sans introduire les invariants de représentation qui nuisent aux preuves de correction. L'idée de l'article en chapitre 2 est très simple : les opérations $\emptyset$, *insert*, *remove*... spécifiées dans IMPL ne sont pas les mêmes que les opérations *abstraites* spécifiées dans MULTI. Elles en sont seulement des réalisations *concrètes*. De même la sorte *Multi* de IMPL est une sorte « concrète » qui n'est pas égale à celle, abstraite, spécifiée dans MULTI. On note alors $\overline{Multi}$, $\overline{\emptyset}$, $\overline{insert}$... les sortes et opérations concrètes de IMPL. On exprime via un « isomorphisme de signatures » la bijection allant des sortes et opérations abstraites vers les sortes et opérations concrètes associées. Cet isomorphisme est noté ρ et est appelé la *représentation* pour des raisons historiques. L'avantage d'une telle approche est que les termes atteints par ρ (plus précisément par son extension canonique aux termes, notée $\overline{\rho}$) sont exactement ceux qui représentent un

élément abstrait. Ainsi les invariants de représentation n'ont pas à être explicités ; on caractérise simplement le « bon domaine » de l'abstraction A par l'image de $\overline{\rho}$. D'autre part, pour les preuves de correction, les junks n'induisent plus aucune inconsistance car au lieu de prouver, par exemple, l'axiome

$$insert(x, M) \subseteq \emptyset \ = \ false$$

on prouve l'axiome

$$\overline{insert}(x, \overline{\rho}(M)) \subseteq \overline{\emptyset} \ = \ false$$

et l'élément non atteignable M_0 précédent n'étant pas de la forme $\overline{\rho}(M)$, il n'est plus une instance acceptable de l'axiome. Ceci sans expliciter les invariants de représentation.

L'approche décrite dans l'article est un peu plus complexe car elle autorise en particulier l'usage d'opérations cachées pour faciliter la spécification de IMPL. De plus divers autres résultats sont prouvés : par exemple que la composition de deux implémentations abstraites correctes fournit des résultats corrects (elle ne forme pas une implémentation abstraite unique, mais il n'y a aucune raison pratique pour que l'on puisse déduire automatiquement une implémentation en une seule étape à partir de deux).

Le défaut de cette approche est qu'il faut néanmoins expliciter la représentation de l'égalité. La question ici est de savoir ce qui nous autorise à considérer deux valeurs concrètes distinctes comme abstraitement identiques. En fait, ceci résulte de l'impossibilité d'*observer* une différence entre ces valeurs concrètes via les opérations fournies. A l'heure actuelle, il semble clair qu'une meilleure approche est de spécifier *ce que l'on observe* au lieu de spécifier la représentation de l'égalité. De plus, nous venons de voir qu'une réalisation correcte d'une spécification abstraite ne satisfait pas nécessairement tous les axiomes. Il en résulte que le principe selon lequel « la classe des modèles satisfaisant une spécification doit représenter toutes les réalisations correctes » n'est pas respecté par la sémantique simple du groupe ADJ que nous avons suivie ici. En fait, on constate que le symbole « = » des spécifications ne doit pas nécessairement être interprété par l'égalité ensembliste, mais seulement par une sorte « d'impossibilité d'observer une différence ». C'est pour cette raison qu'une partie des travaux exposés ici concerne les sémantiques observationnelles des types abstraits algébriques. Dans le cadre de l'implémentation abstraite, nous avons bon espoir que la définition d'une implémentation correcte devienne presque triviale avec de telles sémantiques, via une simple inclusion de modèles (mais ceci n'est pas encore établi).

3 Test de logiciel

Aphorisme : *L'échec d'un test, c'est le succès du testeur.*

Les détails techniques relatifs à cette section sont consignés dans les articles intitulés « *Testing against formal specifications : a theoretical view* » et « *Software testing based on formal specifications : a theory and a tool* » reproduits en chapitres 3 et 4 respectivement.

Même si les étapes de réification (par implémentations abstraites successives ou par une autre approche) ont été poussées assez loin pour que la réalisation d'un logiciel ne pose pas trop de difficultés, il n'en reste pas moins que le produit final obtenu n'est pas un objet formel. Il faut donc encore le vérifier. Une solution possible (peut-être même devrions nous dire nécessaire) est de tester ce produit. Souvent le test est considéré, et pratiqué, d'une manière totalement informelle, si bien qu'on n'a aucune idée de ce qu'il révèle (ou ne révèle pas). C'est sans doute pour cette raison que tant de chercheurs répugnent à étudier, ou même considérer, la phase de test. Notre approche est au contraire motivée par le fait que, au moins à l'heure actuelle,

le développement d'un logiciel, même de taille modeste, requiert une phase de test... Quitte à tester, faisons le proprement.

A la question évidente *que démontre le succès d'un test ?* une réponse non moins évidente est en général *(presque) rien* (i.e. le logiciel n'est pas pour autant correct). Hormis le fait déjà souligné qu'un système logiciel n'est pas un objet mathématique, déduire de cette réponse que le test n'a aucun intérêt serait oublier que la réponse est identique à la question évidente *que démontre l'échec d'une preuve ?* (le logiciel n'est pas pour autant incorrect). De même que l'usage d'une preuve s'avère fructueux lorsque celle-ci réussit, l'usage du test s'avère fructueux lorsque celui-ci échoue. Par conséquent, lors de la sélection d'un jeu de tests, le but visé est de faire échouer le logiciel sous test. La théorie du test est donc en quelque sorte une théorie du contre-exemple. Pour être plus précis, la théorie que nous allons décrire ici est une théorie de la *sélection de jeux de tests* et nous ne considérerons pas les difficultés de mise en œuvre de ces jeux de tests. Toutefois, nous prendrons en compte le problème de la décision du succès ou de l'échec d'un jeu de tests car il a un impact majeur sur la sélection des tests. En effet, il ne faut sélectionner que des tests pour lesquels cette décision est possible.

Nous sommes confrontés à la question suivante : comment se donner le plus de chances de sélectionner un contre-exemple à la correction d'un logiciel ? En pratique on décide qu'un jeu de tests est « bon » lorsque l'on pense que, compte tenu des connaissances que l'on *suppose* raisonnablement avoir sur le logiciel, les tests que l'on a sélectionnés devraient mettre en évidence les *erreurs possibles*. Pour aborder rigoureusement cette question, l'intuition précédente apporte deux indications majeures : on fait des *hypothèses* sur le logiciel, et l'on doit caractériser les erreurs possibles. Puisque l'on dispose d'une spécification algébrique comme référence de correction, l'ensemble des erreurs possibles est aisément caractérisé : c'est la possibilité que le logiciel contredise l'un des axiomes de la spécification. L'idée majeure de notre approche est de s'astreindre à formuler aussi les hypothèses. Un bon jeu de tests doit alors vérifier que *si le logiciel satisfait les hypothèses de test et s'il n'est pas correct* (i.e. s'il ne satisfait pas l'un des axiomes) *alors le jeu de tests échoue*. De manière encore informelle (mais qui sera parfaitement définie un peu plus loin), soit H l'ensemble des hypothèses que l'on fait sur un logiciel P (comme « Programme »), soit $SPEC$ la spécification de référence et soit T le jeu de tests sélectionné, T sera un « bon » jeu de tests si

$$P \models H \quad \wedge \quad P \not\models SPEC \Longrightarrow \text{échec}(T) \tag{1}$$

On rencontre souvent des discussions passionnées à propos des buts du test. Certains affirment, comme nous venons de le faire, que la sélection de jeux de tests a pour but de dégager des erreurs ; d'autres affirment qu'elle a pour but d'établir que le logiciel répond à une « certaine conformité »vis-à-vis de la spécification. Grâce à l'introduction explicite des hypothèses que nous préconisons, ce conflit n'a aucune raison d'être. C'est un faux problème puisque ces deux approches sont strictement équivalentes. En effet, une contraposée de l'implication (1) précédente

$$P \models H \quad \wedge \quad \text{succès}(T) \Longrightarrow P \models SPEC$$

signifie que *sous réserve des hypothèses, le succès du test établit la correction*. Autrement dit, les hypothèses donnent un sens précis à l'expression « une certaine conformité ». Lorsque les hypothèses sont bien choisies, on peut de plus les exploiter en conjonction avec la spécification pour sélectionner automatiquement des jeux de tests. Notons que l'usage d'une spécification *formelle* est crucial ici : une spécification informelle ne peut pas être directement exploitée mécaniquement, ni même servir de base à une approche rigoureuse du test (exactement comme pour la preuve). En tout état de cause, sur un plan méthodologique, il semble préférable de poser les hypothèses d'abord et de sélectionner des jeux de tests seulement ensuite, en accord avec ces hypothèses.

Afin de rester dans un cadre algébrique homogène, nous commencerons par faire une hypothèse « violente » : on supposera que le logiciel sous test, P, définit une algèbre finiment engendrée A_P. La signature sous-jacente à cette algèbre n'est pas très difficile à déterminer en pratique : ce sont les opérations exportées par le programme P considéré. Cette hypothèse est « violente » car elle occulte totalement le fait déjà mentionné qu'un logiciel n'est pas un objet formel a priori. On pourrait penser par la même occasion que notre argument selon lequel le test est nécessaire justement parce que le logiciel n'est pas un objet formel ne tient plus. Il n'en est rien car cette hypothèse simplificatrice n'est faite que pour bâtir une *théorie de la sélection*, et les jeux de tests seront par contre soumis effectivement au système logiciel (dans son environnement global). On parle alors de test *dynamique*, par opposition au test *statique* qui est seulement fondé sur une analyse des sources du logiciel. Notons dès maintenant que si la signature issue du logiciel ne coïncide pas avec celle de la spécification, le logiciel peut d'ores et déjà être considéré incorrect[14]. Nous supposerons donc dorénavant que ces deux signatures coïncident.

A ce stade, nous disposons ainsi d'une spécification algébrique $SPEC$ sur une signature Σ et d'un programme P supposé définir une algèbre A_P élément de $Gen(\Sigma)$. Notons que nous ne savons absolument pas de quelle algèbre il s'agit (sinon il serait inutile de tester le logiciel) la seule hypothèse que nous ayons faite pour le moment est « $A_P \in Gen(\Sigma)$ ». Dans cette section, nous ferons toujours au moins cette hypothèse. Dès lors, toute hypothèse H supplémentaire sera caractérisée par une sous-classe de $Gen(\Sigma)$ que l'on notera par convention $Gen(H)$. De même, par convention, $A_P \models H$ signifiera $A_P \in Gen(H)$. Pour donner un sens rigoureux à l'implication (1) précédente, il reste alors à définir ce qu'est un jeu de tests T et ce que signifie le succès ou l'échec de T.

Rappelons qu'un test peut être vu comme un contre-exemple potentiel à la correction du logiciel par rapport à $SPEC$. Un test est donc en fait une propriété susceptible d'être mise en défaut. A supposer que l'on soit capable de décider de leur succès ou de leur échec, les axiomes de $SPEC$ forment un jeu de tests idéal. En fait, nous partirons de ce jeu de tests idéal et nous l'affinerons jusqu'à ce que la décision succès/échec soit possible. Pour cette raison, un test élémentaire sera simplement une Σ-formule (éventuellement avec variables) et un jeu de tests T sera donc simplement un ensemble de Σ-formules (éventuellement infini). Ceci nous permettra de considérer $T = Ax$, l'ensemble des axiomes de $SPEC$, comme un jeu de tests particulier.

Dès lors, étant donné P, la décision succès/échec peut être définie comme un simple prédicat *partiellement défini* sur l'ensemble des Σ-formules. Ce prédicat est appelé *l'oracle* et est noté O_P. C'est ici que l'on retrouve en quelque sorte « le monde réel »car $O_P(\tau)$ représente à la fois les modalités de soumission du test τ au programme P et la décision de succès ou d'échec : l'algèbre A_P ne joue aucun rôle dans la définition de l'oracle. La réalisation de O_P peut éventuellement mettre en œuvre une instrumentation très élaborée. Cette étape d'oracle est en effet loin d'être triviale. Prenons un exemple très simple pour fixer les idées. Supposons que l'on veuille tester une réalisation des multi-ensembles par des listes (partant de la réification donnée en exemple dans la section 2). On devra essayer de mettre en défaut les axiomes de la spécification MULTI (figure 2, page 10). Par exemple, l'axiome

$$insert(x, insert(y, M)) = insert(y, insert(x, M))$$

pourra donner lieu au test élémentaire

$$insert(x_0, insert(y_0, \emptyset)) = insert(y_0, insert(x_0, \emptyset)) \quad (2)$$

[14]même s'il exporte en fait *plus* d'opérations que celles spécifiées, car on contredirait alors les principes d'encapsulation ; les opérations supplémentaires ne devraient pas être exportées.

où x_0 et y_0 sont deux éléments distincts particuliers. Une fois soumis au programme, ce test résultera en deux listes différentes au sein du programme :

$$(x_0, 1) :: (y_0, 1) :: [\,] \quad \neq \quad (y_0, 1) :: (x_0, 1) :: [\,]$$

Non seulement il n'est pas toujours facile d'accéder à ces listes, mais encore faudra-t'il que O_P soit apte à décider de leur égalité abstraite. En pratique, il est souvent plus judicieux de ne pas sélectionner un tel test, considérant que O_P n'est pas défini sur des formules telles que (2). On le remplacera par exemple par les deux tests suivants

$$insert(x_0, insert(y_0, \emptyset)) \subseteq insert(y_0, insert(x_0, \emptyset)) = true$$

$$insert(y_0, insert(x_0, \emptyset)) \subseteq insert(x_0, insert(y_0, \emptyset)) = true$$

qui sont beaucoup plus facilement *observables* et décidables, et seront donc dans le domaine de définition de O_P.

Au départ du processus de sélection, nous disposons du programme P sous test et d'une spécification $SPEC$, qui, eux, ne varieront pas. Nous cherchons à faire évoluer trois composantes de front : un ensemble H d'hypothèses sur A_P, un jeu de tests T et un oracle O_P. A priori, au début de ce processus, H est réduit à « $A_P \in Gen(\Sigma)$ » (puisque la seule chose que l'on connaisse sur le programme est sa signature), T est égal à l'ensemble des axiomes de $SPEC$ (qui est la liste exhaustive des « erreurs » potentielles de P), et l'oracle O_P est indéfini (on ne sait pas quelles sont les propriétés trivialement décidables). Les triplets (H, T, O) sont appelés des *contextes de test* et le triplet de départ que nous venons de décrire est appelé le *contexte canonique*, noté par la suite (H_0, T_0, O_0). La propriété que l'on cherche à atteindre est que, d'une part T soit fini[15] et inclus dans le domaine de définition de O_P, et d'autre part la condition suivante soit satisfaite :

$$A_P \models H \ \wedge \ O_P(T) \implies A_P \models SPEC$$

ou même mieux :

$$A_P \models H \ \wedge \ O_P(T) \iff A_P \models SPEC \tag{3}$$

La véritable difficulté est que ces équivalences n'ont de sens que si T est inclus dans le domaine de définition de O_P.

On remarque que le contexte canonique (H_0, T_0, O_0) décrit précédemment possède la propriété remarquable suivante :

$$A_P \models H_0 \ \wedge \ A_P \models T_0 \iff A_P \models SPEC \tag{4}$$

Malheureusement, T_0 n'est pas inclus dans le domaine de définition de O_0, et c'est tout le travail de la phase de sélection que d'obtenir cette propriété. Ceci sans corrompre la propriété (4), en assurant de plus que $O(T)$ soit équivalent à $A_P \models T$ et que le jeu de tests T obtenu soit fini. On assurera ainsi la propriété (3). Souvent, ceci impose de faire disparaître les variables de T_0. En effet, pour la plupart des programmes, seuls les termes sans variables sont exécutables. A fortiori O_P ne peut généralement être défini que sur des formules closes.

Un peu de *terminologie* : la propriété (3) peut être scindée en trois.

- $A_P \models H \ \wedge \ O_P(T) \implies A_P \models SPEC$
  qui, via un petit exercice de contraposée, signifie « sous réserve des hypothèses, le jeu de tests et l'oracle détectent tout programme incorrect ». Cette propriété est appelée la *validité* du contexte de test[16]. On constate clairement que pour préserver la validité d'un contexte de test tout en diminuant la portée de T (par exemple supprimer ses variables en choisissant des instances particulières) il faut augmenter les hypothèses de test.

[15] et de taille raisonnable...

[16] pour des raisons historiques... à l'heure actuelle le mot *complétude* serait sans doute mieux choisi.

– $A_P \models SPEC \Longrightarrow O_P(T)$
qui signifie que le jeu de tests et l'oracle ne rejettent aucun programme correct. Cette propriété est appelée le *non biais* du contexte de test.
– enfin une propriété plus discutable :
$A_P \models SPEC \Longrightarrow A_P \models H$
qui signifie que tous les programmes corrects répondent aux hypothèses de test. Cette propriété est appelée la *conservativité* de H. Elle est discutable car, a priori, rien n'interdit de connaître par ailleurs certaines particularités du programme sous test. On peut alors parfaitement en déduire que A_P ne peut pas appartenir à certaines classes de réalisations correctes et exploiter cette connaissance pour formuler des hypothèses plus fines.
Puisque nous étudions ici une approche purement « boîte noire », toutes nos hypothèses seront conservatives. Dans le cas contraire, l'équivalence (3) serait avantageusement remplacée par

$$A_P \models H \Longrightarrow [\, O_P(T) \iff A_P \models SPEC \,]$$

Les articles sur le test des chapitre 3 et 4 définissent un *préordre d'affinement* sur les contextes de test. Ce préordre traduit des conditions suffisantes raisonnables pour préserver la propriété (4) du contexte canonique, tout en convergeant vers un jeu de tests fini, inclus dans le domaine de définition de l'oracle et de telle sorte que $O(T)$ soit équivalent à $A_P \models T$. Pour ce faire, il faut augmenter les hypothèses de test. Ainsi, dire que (H_2, T_2, O_2) est un affinement de (H_1, T_1, O_1) traduit intuitivement les trois conditions suivantes :
– L'hypothèse H_2 est plus forte que l'hypothèse H_1.
– Ajouter des hypothèses de test autorise à réduire la portée de T, mais il faut toutefois préserver la validité. Ceci est exprimé par la condition que *sous les hypothèses les plus fortes*, c'est-à-dire H_2, T_2 révèle au moins autant de programmes incorrects que T_1 (i.e. si $A_P \not\models T_1$ alors $A_P \not\models T_2$). Notons qu'ici l'oracle n'intervient pas encore, on met en jeu $A_P \models T_i$ et non pas $O_i(T_i)$.
– Enfin O_2 étend O_1. C'est-à-dire que le domaine de définition de O_2 contient au moins celui de O_1, et ils coïncident sur le domaine de définition de O_1.

Divers théorèmes résultent de la définition de ce préordre d'affinement. On peut démontrer entre autres que le préordre d'affinement préserve la validité. Il se trouve de plus qu'on peut donner un contexte de test majorant qui assure que l'oracle coïncide avec la relation de satisfaction là où il est défini (i.e. $O_P(\tau) \iff A_P \models \tau$). Ceci nous assure que les contextes de test obtenus par affinements successifs préservent les bonnes propriétés (validité et non biais).

Donnons maintenant un exemple d'affinements successifs de contextes de test. Partons de la spécification MULTI (figure 2, page 10) et considérons l'hypothèse minimale déjà mentionnée ($A_P \in Gen(\Sigma)$), le jeu de tests contenant les axiomes de MULTI, et l'oracle indéfini $O_P = undef$. Notons $\mathcal{C}_0 = (\, H_0 \,,\, T_0 \,,\, undef \,)$ ce contexte canonique. Nous allons voir comment obtenir un jeu de tests pertinent en ajoutant des hypothèses bien choisies ; d'abord des hypothèses de « régularité » puis « d'uniformité » et enfin « d'oracle ». Nous considérerons plus spécifiquement l'axiome τ_0 suivant

$$insert(x, insert(y, M)) = insert(y, insert(x, M))$$

On suppose que P ne peut exécuter que des termes clos, et que par conséquent il n'y a aucune chance pour que O_P puisse être défini sur des formules contenant des variables. Commençons donc par supprimer la variable M de τ_0. En pratique, un testeur se restreindra souvent à instancier M par le multi-ensemble vide, puis par un multi-ensemble de cardinal 1, de cardinal 2, etc ; jusqu'à un certain cardinal qu'il considérera suffisant. L'hypothèse sous-jacente à cette pratique s'appelle une *hypothèse de régularité* pour τ_0. Elle fait appel à une « fonction d'intérêt » α qui associe un entier naturel à chaque terme de sorte $Multi$ ne contenant que des constructeurs (ici $\emptyset$ et

insert) et des variables de sortes utilisées (ici *Elem*). Pour l'exemple que nous décrivons, α est simplement le nombre d'occurrences de *insert*. L'hypothèse est alors notée $Régul_n^\alpha(\tau_0, M)$, et $A_P \models Régul_n^\alpha(\tau, M)$ signifie par définition :

$$(\forall M \in A_{P,\Omega})(\ \alpha(M) \leq n \Longrightarrow \tau\ ) \implies (\forall M \in A_{P,s})(\ \tau\ )$$

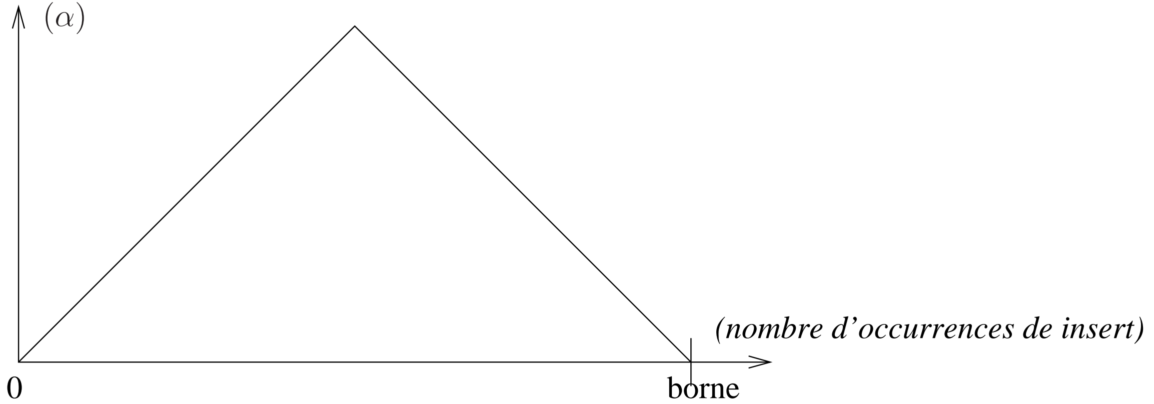
où s est la sorte de M (ici $s = Multi$) et Ω est l'ensemble des générateurs de cette sorte (ici $\emptyset$ et *insert*) ; $A_{P,\Omega}$ est donc l'ensemble des valeurs de $A_{P,s}$ atteignables par $\emptyset$ et *insert*. Considérons maintenant cette hypothèse de régularité avec $n = 1$ pour éviter de retranscrire trop de cas de tests. Ceci nous conduit à un contexte de test $\mathcal{C}_1$ où l'oracle n'a pas changé, l'hypothèse H_1 est égale à la conjonction de H_0 et $Régul_1^\alpha(\tau_0, M)$, et où le jeu de tests T_1 est obtenu à partir de T_0 en remplaçant τ_0 par les deux axiomes suivants :

$$insert(x, insert(y, \emptyset)) = insert(y, insert(x, \emptyset))$$

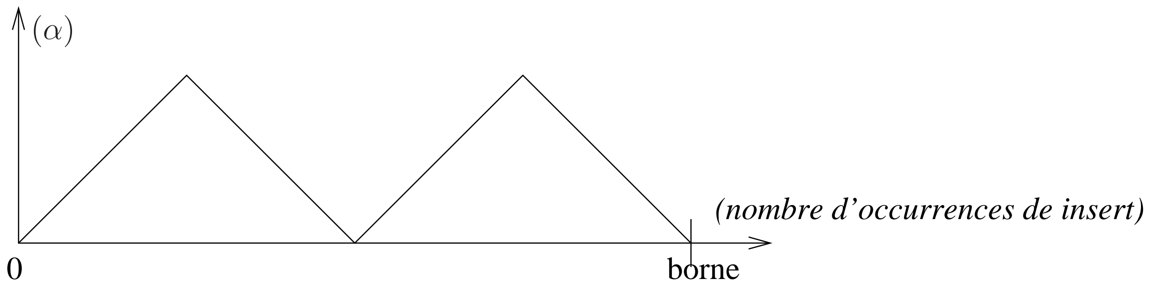
$$insert(x, insert(y, insert(z, \emptyset))) = insert(y, insert(x, insert(z, \emptyset)))$$

Il n'est pas difficile de démontrer que $\mathcal{C}_1$ affine $\mathcal{C}_0$. Nous constatons que toutes les variables de sorte $Multi$ ont disparu (en un nombre fini d'étapes similaires, on peut obtenir le même résultat pour des axiomes contenant plusieurs variables de sorte $Multi$, et pour tous les axiomes de T_0).

Remarquons que les hypothèses de régularité sont en fait très souples car la fonction d'intérêt α peut être définie de manière arbitraire. Une lacune de la fonction que nous avons choisie est que seuls des ensembles de petite taille sont testés. Si l'on veut tester des structures bornées, il est préférable de choisir des fonctions d'intérêt retenant des tests autour des bornes (à savoir des multi-ensembles de faible taille et des multi-ensembles « presque pleins »). La fonction d'intérêt α devra alors prendre de faibles valeurs vers les bornes, afin que le critère $\alpha(M) \leq n$ retienne aussi des multi-ensembles presque pleins. On peut, par exemple, choisir une fonction d'intérêt ayant l'allure suivante :



Si l'on veut retenir aussi quelques tests « au milieu », on peut préférer :



Toutes ces fonctions d'intérêt s'expriment assez facilement via un usage judicieux de valeurs absolues. Par exemple la première est obtenue via la formule[17]

$$\alpha(M) = borne/2 - abs(nbocc(M) - borne/2)$$

où $nbocc(M)$ est le nombre d'occurrences de *insert* dans M. Des recherches sur le test de structures bornées, et plus généralement comportant des traitements d'exceptions, sont en cours dans la thèse de Pascale Le Gall[18].

Il faut maintenant s'affranchir des variables de sorte *Elem*, par exemple x, y et z de l'axiome τ_1 issu de l'affinement précédent :

$$(\tau_1) \qquad insert(x, insert(y, insert(z, \emptyset))) = insert(y, insert(x, insert(z, \emptyset)))$$

En pratique, un testeur se contente généralement de choisir au hasard une valeur pour chacune de ces variables, et engendre donc un test par cardinal retenu. Intuitivement, il considère par là même que les opérations mises en jeu dans τ_1 (ici *insert*) ont un comportement qui ne dépend pas de la valeur insérée, c'est-à-dire n'exploitent pas la valeur particulière de x, y ou z lors de leur exécution. Cette hypothèse s'appelle une *hypothèse d'uniformité* relativement aux variables de τ_1. Une telle hypothèse est notée $Unif(\tau, x)$ et par définition $A_P \models Unif(\tau, x)$ signifie :

$$(\exists x \in A_{P,s})(\ \tau\ ) \implies (\forall x \in A_{P,s})(\ \tau\ )$$

où s est la sorte de x (ici $s = Elem$). Les hypothèses d'uniformité peuvent paraître très fortes. En fait, elles sont moins fortes que les hypothèses de régularité : il est souvent possible de vérifier une hypothèse d'uniformité, au moins partiellement, en s'assurant qu'aucun prédicat mettant en jeu les variables considérées n'est exploité dans l'algorithme réalisant *insert*.

En trois étapes évidentes, l'uniformité nous autorise à introduire des valeurs particulières x_0, y_0 et z_0 et à affiner le contexte de test $\mathcal{C}_1$ en un contexte de test $\mathcal{C}_2$ où l'oracle n'a pas changé, l'hypothèse H_2 est égale à la conjonction des hypothèses H_1, $Unif(\tau_1, x)$, $Unif(\tau_1[x \leftarrow x_0], y)$ et $Unif(\tau_1[x \leftarrow x_0][y \leftarrow y_0], z)$, et où enfin T_2 est obtenu à partir de T_1 en remplaçant l'axiome τ_1 par la formule sans variable τ_2 suivante :

$$(\tau_2) \qquad insert(x_0, insert(y_0, insert(z_0, \emptyset))) = insert(y_0, insert(x_0, insert(z_0, \emptyset)))$$

Nous n'avons développé ici qu'un exemple d'axiome non conditionnel. Les hypothèses d'uniformité que nous avons utilisées sont en fait trop fortes pour traiter des axiomes conditionnels. Dans ce dernier cas, il est préférable de choisir des instances des variables qui satisfont les prémisses des axiomes. Remarquons que dès lors, pour sélectionner automatiquement des jeux de tests, des procédures de résolution équationnelles sont très utiles pour résoudre les prémisses. C'est en particulier le cas des listes triées, où x_0, y_0 et z_0 devraient être choisis en ordre décroissant par exemple. On peut démontrer alors qu'il faut appliquer les hypothèses de régularité *avant* les hypothèses d'uniformité. Ceci est intuitivement bien clair : pour des listes d'entiers naturels triées par ordre strictement décroissant (donc sans redondance), on serait bien embarrassé pour choisir une instance de z si l'on avait au préalable sélectionné $x_0 = 1$ et $y_0 = 0$. Ceci étant précisé, cette technique peut être appliquée à tous les axiomes de T_1 et nous obtenons ainsi un jeu de tests fini ne contenant plus aucune variable. Notons $\mathcal{C}_3 = (H_3, T_3, undef)$ le contexte de test ainsi obtenu[19].

[17] On suppose ici que la borne est paire pour simplifier.

[18] que j'encadre actuellement.

[19] Notons qu'en fait un tel contexte de test $\mathcal{C}_3$ ne peut être obtenu via les techniques mentionnées que si les axiomes de départ ne contiennent que des variables universellement quantifiées.

Tous les tests de T_3 peuvent a priori être exécutés par le programme sous test, puisqu'ils ne contiennent aucune variable et que le programme fournit toutes les opérations de la signature. Nonobstant, l'oracle reste indéfini.

Il est généralement très facile de décider du succès ou de l'échec d'une formule ne mettant en cause que des sortes « basiques » (par exemple les booléens, les entiers, les chaînes de caractères...). Ceci résulte du fait que ces sortes sont généralement *observables* (on comprend dès lors pourquoi l'oracle est l'une des motivations pour étudier l'observabilité, cf. section 4). En pratique on peut donc fournir un oracle pour de telles formules. Notons que pour notre approche fondée sur les spécifications algébriques, ceci revient simplement à être capable de décider du succès ou de l'échec d'une *égalité* au sein de ces types basiques. Les tables de vérité des connecteurs logiques (implication, conjonction, etc) font le reste. On peut alors étendre le domaine de définition de O_P à ces formules. Pour des raisons techniques, on doit ajouter l'hypothèse suivante pour chaque sorte observable[20] s, qui définit en fait ce que signifie « observable » pour notre approche, il s'agit d'une simple observation de sortes :[21]

$$(\forall t_1,\ t_2 \in T_{\Sigma,s})(\ O_P(t_1 = t_2) \Longleftrightarrow A_P \models (t_1 = t_2)\ )$$

Une partie des tests de T_3 peut donc d'ores et déjà être soumise au logiciel sous test, par exemple le test suivant :

$$insert(x_0, \emptyset) \subseteq \emptyset = false$$

qui est issu de l'avant dernier axiome de MULTI (figure 2, page 10) ; il appartient au domaine de définition de O_P. Plus généralement, tous les axiomes définissant l'inclusion « $\subseteq$ » peuvent être testés. Par contre il n'en est pas de même pour le test τ_2 (cf. la discussion au début de cette section, page 15). Soient t_2 et t'_2 les deux membres de τ_2 (i.e. τ_2 est le test $t_2 = t'_2$). Ce qui justifie que les représentations de t_2 et t'_2, bien qu'elles soient distinctes, fournissent des valeurs abstraitement égales, c'est qu'aucune suite d'opérations conduisant à un résultat observable, appliquée à t_2 et à t'_2 respectivement, ne permet de les distinguer. Avec une sémantique observationnelle bien choisie (voir section 4 plus loin), ceci conduirait à pouvoir remplacer, sans aucune hypothèse supplémentaire, le test τ_2 par l'ensemble de tests suivant :

$$\begin{gathered}
t_1 \subseteq \emptyset = t_2 \subseteq \emptyset \\
t_1 \subseteq insert(x_0, \emptyset) = t_2 \subseteq insert(x_0, \emptyset) \\
t_1 \subseteq insert(y_0, insert(x_0, \emptyset)) = t_2 \subseteq insert(y_0, insert(x_0, \emptyset)) \\
\dots \\
\emptyset \subseteq t_1 = \emptyset \subseteq t_2 \\
insert(x_0, \emptyset) \subseteq t_1 = insert(x_0, \emptyset) \subseteq t_2 \\
insert(y_0, insert(x_0, \emptyset)) \subseteq t_1 = insert(y_0, insert(x_0, \emptyset)) \subseteq t_2 \\
\dots \\
insert(x_0, t_1) \subseteq \emptyset = insert(x_0, t_2) \subseteq \emptyset \\
insert(y_0, insert(x_0, t_1)) \subseteq \emptyset = insert(y_0, insert(x_0, t_2)) \subseteq \emptyset \\
\dots \\
insert(x_0, t_1) \subseteq insert(y_0, \emptyset) = insert(x_0, t_2) \subseteq insert(y_0, \emptyset) \\
\text{et bien d'autres...}
\end{gathered}$$

Malheureusement ce jeu de tests est infini. Même si l'on remplace les deux premiers groupes d'axiomes par ($t_1 \subseteq M = t_2 \subseteq M$) et ($M \subseteq t_1 = M \subseteq t_2$) puis que l'on applique les techniques précédentes pour supprimer la variable M, cette méthode ne couvre pas les autres groupes de

[20] i.e. les sortes basiques déjà mentionnées.

[21] Noter que, O_P étant donné, il s'agit bien d'une hypothèse sur l'algèbre A_P.

tests. Il en résulte que *le passage à une sémantique observationnelle des spécifications ne suffit pas pour résoudre l'oracle.* Pour l'exemple que nous considérons ici, comme nous l'avons déjà mentionné, il *semble* raisonnable de remplacer le test τ_2 par les deux tests τ_3 et τ_4 suivants :

$$t_1 \subseteq t_2 = true \quad \text{et} \quad t_2 \subseteq t_1 = true$$

Pourquoi croyons-nous ces deux tests suffisants ? En fait, ceci résulte du théorème suivant :

$$t_1 \subseteq t_2 = true \ \wedge \ t_2 \subseteq t_1 = true \Longleftrightarrow t_1 = t_2$$

qui peut être démontré sans grande difficulté *à partir des axiomes de* MULTI. C'est là que le bât blesse. Nous sommes justement en train de tester le logiciel (non la spécification) pour savoir si, compte tenu des hypothèses, il satisfait ou non la spécification. Donc nous ferions une pétition de principe en remplaçant purement et simplement τ_2 par ces deux tests. Le contre-exemple décrit dans la section 9 du chapitre 3 démontre que ce vice de forme ne relève pas seulement de « scrupules de théoricien ». Même en pratique, les risques de pétition de principe induits par un oracle mal conçu sont énormes. Pourtant, puisque nous venons de voir que l'inclusion pouvait être testée au préalable, *faire l'hypothèse* que τ_2 équivaut à ces deux tests pour le logiciel considéré semble une approche raisonnable. Cette hypothèse d'oracle s'exprime comme suit : *pour tous M_1 et M_2 de A_P, M_1 et M_2 de sorte Multi, on a*

$$A_P \models (M_1 = M_2) \Longleftrightarrow \left\{ \begin{array}{c} A_P \models (M_1 \subseteq M_2 = true) \\ \text{et} \\ A_P \models (M_2 \subseteq M_1 = true) \end{array} \right\}$$

Après avoir exprimé de telles équivalences pour chaque sorte non observable de la spécification, en tant qu'hypothèses de test, on peut réduire le jeu de tests à un ensemble fini de tests appartenant au domaine de définition de l'oracle (par étapes d'affinement successives). Là encore, un apport majeur de notre théorie est d'obliger à formuler ces hypothèses pour préserver le préordre d'affinement. De cette façon nous pouvons donc sélectionner des jeux de tests pertinents pour toute spécification algébrique (les hypothèses fournissant justement une bonne mesure de cette « pertinence »).

Mentionnons pour terminer que ces travaux ont été développés sous l'impulsion de Marie-Claude Gaudel et que Bruno Marre a développé dans le même temps un système qui engendre automatiquement des jeux de tests à partir d'une spécification algébrique. Ce système est fondé sur des techniques de programmation logique équationnelle avec contraintes. Il suffit de lui fournir la spécification et chaque niveau n de régularité souhaité. Il suit exactement la théorie de sélection de jeux de tests décrite ici sauf qu'il ne prend pas encore en compte la phase d'oracle décrite ci-dessus. Les apports réciproques entre la conception du système et celle de la théorie ont été constants. Des développements futurs certainement prometteurs seront d'étendre le système à cette phase d'oracle ainsi qu'au traitement des « exception-spécifications », pour prendre en compte en particulier le test de structures bornées.

4 Sémantiques de l'observabilité

> Aphorisme : *Les modèles intéressants d'une spécification algébrique ne sont pas ceux qui satisfont ses axiomes.*

Les détails techniques relatifs à cette section sont consignés dans l'article intitulé « *Proving the correctness of algebraically specified software : modularity and observability issues* » reproduit en chapitre 5.

Nous avons mentionné qu'une spécification doit exprimer le « quoi » et non le « comment ». Le but que nous cherchons à atteindre pour faciliter la validation est d'autoriser un style de spécification *clair*, *concis* et très *abstrait*. Sinon il serait tout aussi simple d'écrire directement un prototype dans un langage de haut niveau, comme ML par exemple. Or il s'avère que lorsque l'on spécifie selon une sémantique classique, on est souvent conduit à exprimer « trop » de propriétés. Les sémantiques observationnelles permettent de spécifier « moins » de propriétés, et de ne retenir que le « quoi » pertinent. De plus, motivés par les sections précédentes sur la vérification, nous devons assurer dans le même temps une sémantique bien choisie, telle que la classe de modèles qu'elle définit représente *toutes* les réalisations acceptables de la spécification (et autant que possible celles-là seulement).

Alors que les sections 2 et 3 se situaient au sein des étapes de réalisation et de vérification d'un logiciel, les sections 5 et 6 concerneront plutôt les étapes de spécification et de validation d'une spécification. Cette section 4 se situe à la croisée des deux en ce sens que l'observabilité peut simplifier, comme nous l'avons mentionné, l'étape de vérification et qu'elle permet surtout un style d'écriture des axiomes très naturel, facilitant par là même la validation.

Nous avons vu en section 2 que la sémantique « classique » (celle décrite par le groupe ADJ [GTW78]) ne couvre pas toutes les réalisations acceptables. Par exemple elle n'admet aucun modèle dont les éléments de type $Multi$ soient des listes de couples (comme décrit en page 11). La raison en est que, considérant par exemple l'axiome

$$insert(x, insert(y, M)) = insert(y, insert(x, M))$$

les deux listes correspondantes ne sont pas égales. Cette réalisation doit pourtant clairement être considérée comme correcte, ce qui justifie l'aphorisme énoncé au début de cette section. Plus sérieusement, cet exemple démontre qu'une bonne sémantique ne doit pas nécessairement interpréter le symbole « = » apparaissant dans la syntaxe d'un axiome comme l'égalité de la théorie des ensembles. Ici ce symbole d'égalité veut plutôt dire, comme disent souvent les physiciens, que *tout se passe comme si*[22] les deux membres étaient égaux. En fait deux listes permutations l'une de l'autre peuvent être considérées comme égales en tant que multi-ensembles uniquement parce qu'on ne dispose que d'une observation restreinte de ces listes via les opérations autorisées sur les multi-ensembles. En l'occurrence, ce que nous admettons implicitement est que la seule observation autorisée est le prédicat d'inclusion (figure 2, page 10). Si l'on s'autorisait à observer directement les résultats fournis par l'opération $insert$ par exemple, alors cette réalisation ne serait pas correcte car nous pourrions « voir » que les deux membres de l'équation précédente diffèrent.

Fidèles à notre approche, nous en concluons qu'il faut avant toute chose *spécifier formellement* ce qui est, ou n'est pas, observable. Il existe principalement quatre niveaux de pouvoir d'expression pour restreindre l'observation :

- On peut spécifier que seulement certaines *sortes* sont observables. On distingue alors un sous ensemble S_{Obs} de l'ensemble S des sortes de la spécification. C'est par exemple l'approche sous-jacente que nous avons utilisée pour traiter l'oracle à la fin de la section 3.
- On peut spécifier que seulement certaines *opérations* rendent des résultats observables. On distingue alors un sous-ensemble Σ_{Obs} de l'ensemble Σ des opérations de la signature.
- On peut spécifier que seulement certains *termes* avec variables sont observables. On décrit alors, par une méthode appropriée, un sous-ensemble (éventuellement infini) T_{Obs} de l'algèbre libre des termes avec (une ou plusieurs) variable(s) $T_\Sigma(X)$. Cette approche couvre en fait largement tous les exemples de spécification que l'on pourrait avoir envie d'écrire, mais définir une sémantique appropriée dans ce cas pose de nombreuses difficultés.

[22] A mon avis, cette locution est un des points caractéristiques des sciences *expérimentales*.

– On peut enfin spécifier que seulement certaines *formules* sont observables. On décrit alors, par une méthode appropriée, un sous-ensemble (éventuellement infini) Φ_{Obs} de l'ensemble des Σ-formules avec variables Φ_Σ. Il est là encore difficile de définir une sémantique appropriée, mais en contre-partie sa généralité en fait une référence permettant d'unifier bon nombre d'autres approches. Par exemple le domaine de définition de l'oracle défini à la fin de la section 3 est un ensemble de formules observables, bien que nous nous soyons contentés d'une observation fondée sur les sortes.

Une *spécification observationnelle* est alors une spécification classique munie d'un ensemble d'observations choisi parmi les quatre approches décrites ci-dessus. Une étude comparative partielle de ces diverses approches est fournie dans « *Observational approaches in algebraic specifications : a comparative study* », article[23] écrit en collaboration avec Michel Bidoit et Teodor Knapik (non reproduit dans ce document). Pour une bonne partie des exemples de spécification observationnelle, une approche fondée sur l'observation des sortes est suffisante, mais il existe plusieurs cas où elle est insuffisante. Pour ces cas, une approche purement fondée sur l'observation d'opérations est souvent elle aussi insuffisante. Nous allons décrire plus loin une théorie de l'observabilité qui combine une observation de termes et de sortes. Nous verrons qu'elle permet de décrire des observations très élaborées sans pour autant soulever les difficultés rencontrées pour l'observation des termes.

Il faut ensuite définir une *sémantique observationnelle* pour les spécifications observationnelles, c'est-à-dire définir quelles Σ-algèbres seront acceptées comme satisfaisant la spécification compte tenu de l'observabilité spécifiée. Il existe deux approches :

– On peut dire qu'un modèle est acceptable si, compte tenu des observations possibles, il se comporte comme un modèle « classique » de la spécification.
– On peut dire qu'un modèle est acceptable si, compte tenu des observations possibles, il se comporte comme si tous les axiomes de la spécification étaient satisfaits.

Contrairement aux apparences, ces deux sémantiques ne sont pas équivalentes. Le chapitre 5 contient un exemple de spécification observationnelle[24] pour laquelle la première approche n'accepte aucun modèle (i.e. est inconsistante), alors que la seconde contient au moins un modèle. Nous allons de plus constater qu'il est plus facile de spécifier « ce qui est observable » via la seconde approche que via la première. Intuitivement, la seconde approche est plus « précise » que la première en ce sens qu'elle traite directement les axiomes de la spécification. Ceci permet d'écrire des spécifications à la fois plus claires et plus concises.

Définissons rapidement la première sémantique. Nous nous placerons dans le cadre de l'observation de formules, qui est la plus générale pour cette sémantique[25].

Définition 1 : Soit $SPEC = (S, \Sigma, Ax, \Phi_{Obs})$ une spécification observationnelle.

Soient A et B deux Σ-algèbres. Elles seront dites *observationnellement équivalentes* si et seulement si pour toute Σ-formule $\varphi \in \Phi_{Obs}$ on a :

$$A \models \varphi \Longleftrightarrow B \models \varphi$$

De plus, une Σ-algèbre A satisfait observationnellement $SPEC$ si et seulement s'il existe une Σ-algèbre B observationnellement équivalente à A et telle que $B \models Ax$ (au sens classique).

Nous ne précisons pas ici quelle est la forme des axiomes ; en fait cette définition est « institution independent ».

[23] Rapport de recherche du LIENS 91-6, Ecole Normale Supérieure, Paris, Avril 1991.
[24] un peu pathologique, il faut bien l'admettre ; c'est pourquoi nous ne le reproduisons pas ici.
[25] Ceci est démontré dans l'article mentionné plus haut, non inclus dans ce document.

Considérons maintenant une spécification des ensembles d'entiers naturels munie d'un énumérateur, par exemple celle donnée en figure 4. On suppose données par ailleurs les spécifications

---

4— Ensembles d'entiers avec énumérateur

Spec « SET ».

S : $\{Set\}$

Σ : $\emptyset$ $:\to Set$

$insert$: $Nat \times Set \to Set$

$\in$: $Nat \times Set \to Bool$

$enum$: $Set \to List$

Ax : $insert(n, insert(m, E)) = insert(m, insert(n, E))$

$insert(n, insert(n, E)) = insert(n, E)$

$n \in \emptyset = false$

$n \in insert(n, E) = true$

$eq(n, m) = false \Longrightarrow n \in insert(m, E)) = n \in E$

$redundant(enum(E)) = false$

$occurs(n, enum(E)) = n \in E$

BOOL, NAT et LIST. On suppose aussi que LIST est munie des opérations $occurs$ et $redundant$ qui déterminent respectivement si un entier n apparaît dans une liste l et si une liste est redondante. Il nous reste à spécifier l'ensemble des formules observables pour cette spécification. Nous supposerons que les entiers et les booléens sont observables, *mais* nous supposerons que les résultats fournis par *enum* ne peuvent donner lieu qu'à une observation réduite. Intuitivement, si un module réalisant SET est utilisé au sein d'un logiciel, *enum* pourra être utilisé comme une opération *interne* à ce logiciel, mais ne devra pas être directement accessible de l'extérieur. En effet, dans la mesure où l'on a spécifié abstraitement des ensembles et non des listes, on ne veut pas que, vu de l'extérieur, un ordre quelconque puisse être privilégié dans un ensemble. Il est alors tentant d'en déduire que les formules observables Φ_{Obs} que nous devons choisir sont celles dans lesquelles *enum* n'apparaît pas. Malheureusement, dans ce cas, les deux derniers axiomes ne servent à rien car ils ne pourront jamais être observés. Nous sommes donc obligés de les inclure dans Φ_{Obs}. Pour résoudre ce problème, on peut *de manière ad hoc*, dans cet exemple, observer les formules de la forme $(t_1 = t_2)$ où t_1 et t_2 sont des termes de sorte Nat ou $Bool$ ne contenant soit aucune occurrence de *enum*, soit seulement des occurrences de la forme $redundant(enum(E))$ ou $occurs(n, enum(E))$. (Remarquons que ceci est parfaitement arbitraire d'un point de vue de spécification, et est uniquement motivé par des raisons techniques.)

Parmi les algèbres A qui satisfont la spécification SET au sens classique, on peut mentionner par exemple celles pour lesquelles A_{Set} est l'ensemble des ensembles finis d'entiers naturels et telles que $enum(E)$ énumère E en ordre croissant. On peut aussi mentionner celles qui l'énumèrent en ordre décroissant, ou bien les éléments pairs par ordre croissant puis les impairs par ordre décroissant, etc. Pour ces algèbres classiques, l'insertion est toujours permutable, et l'insertion multiple d'un même élément est toujours idempotente. Le point important ici est que par conséquent, étant donné un ensemble E, $enum(E)$ ne dépend jamais de l'ordre d'insertion des éléments dans cet ensemble[26]. Considérons maintenant la réalisation de SET obtenue simplement en représentant un ensemble par la liste de ses éléments, *insert* étant réalisé par *cons*[27] si l'élément à insérer n'est pas déjà dans la liste, l'identité sinon. Il semble alors naturel de réaliser

[26]En effet, toutes les opérations de la signature sont déterministes par définition ; des travaux considérant une opération *enum* non-déterministe sont en cours dans la thèse de Anne Deo-Blanchard, que j'encadre actuellement.

[27]aussi noté « : : » en section 2.

enum par une simple coercion du type *Set* dans le type *List*. Ce modèle, que l'on notera A, ne satisfait pas la spécification SET avec la sémantique classique, puisque $enum(E)$ dépend de l'ordre d'insertion des éléments. Cependant, étant donnée une algèbre B satisfaisant SET au sens classique (par exemple énumérant les éléments en ordre croissant), il est clair que toute formule de Φ_{Obs} est valide dans A si et seulement si elle l'est dans B. En effet, en ce qui concerne les formules ne faisant pas intervenir *enum*, et compte tenu de la signature de SET, les seules observations possibles d'un ensemble mettent en jeu des termes de la forme $e \in E$, pour lesquels le résultat est identique dans A ou dans B. En ce qui concerne les formules faisant intervenir *enum*, les seules observations retenues sont celles qui mettent en jeu des termes de la forme $redundant(enum(E))$ ou $occurs(e, enum(E))$, pour lesquels là encore le résultat est identique dans A ou dans B. Il en résulte que A et B sont observationnellement équivalentes par rapport à Φ_{Obs}. A est donc un modèle observationnel de (SET, Φ_{Obs}). Si par contre nous considérons un modèle C où *insert* est toujours réalisé par *cons*, que l'élément soit présent ou non, alors l'axiome $(redundant(enum(E)) = false)$, qui appartient à Φ_{Obs}, est valide dans B mais pas dans C. Donc B et C ne sont pas observationnellement équivalents. On voit donc bien comment l'équivalence observationnelle permet d'étendre judicieusement la classe des modèles acceptables d'une spécification. Elle accepte des modèles qui ne satisfont pas tous les axiomes selon l'égalité de la théorie des ensembles (par exemple $insert(n, insert(m, E)) = insert(m, insert(n, E))$), mais sont « acceptables à observation près ».

Nous ferons deux critiques à cette approche par équivalence observationnelle de modèles.

- D'abord une critique mineure : cette sémantique repose sur une notion de satisfaction classique qui est seulement étendue a posteriori via l'équivalence observationnelle. Ceci conduit à une théorie relativement hétérogène, dans laquelle la relation de satisfaction observationnelle d'un axiome par une algèbre n'est pas définie (i.e. « $A \models \varphi$ observationnellement » n'a pas de sens). Elle ne peut être définie que pour une spécification observationnelle complète ($A \models SPEC$ est exprimé par la définition 1). En conséquence, cette approche ne permet pas de définir une *institution* des spécifications observationnelles (voir section 6).
- La seconde critique est plus lourde de conséquences sur le plan pratique : choisir Φ_{Obs} judicieusement est une tâche complexe. Par exemple pour *enum*, il peut être judicieux que vues de l'extérieur, les opérations « commutatives » puissent être appliquées à l'énumération d'un ensemble. C'est le cas de *redundant* et de *occurs*, mais ce n'est pas le cas de *car* ou *cdr* par exemple. Selon la signature des listes considérée, on doit alors admettre l'observation de $length(enum(E))$ par exemple. Ceci signifie qu'il faut *préalablement* prouver diverses propriétés des opérations de la signature avant de spécifier les observations. Le processus de spécification, et encore plus celui de validation, s'en trouvent compliqués d'autant. Il faudrait un moyen de spécifier que *enum* interdit toute observation par défaut, mais, comme nous l'avons déjà mentionné, ceci n'est pas possible à cause des deux derniers axiomes de la spécification.

Nous allons voir maintenant qu'une sémantique fondée sur l'autre approche, qui étudie l'observabilité axiome par axiome, n'est pas sujette à ces deux critiques car elle est fondée sur une notion de « contextes observables »qui est indépendante des axiomes de la spécification.

Le point clef pour ce second type de sémantique est de définir la satisfaction d'une égalité observationnelle $t_1 = t_2$ dans le cas où t_1 et t_2 ne sont pas observables. Dans le cadre observationnel, ceci signifie qu'aucune suite d'opération conduisant à un résultat observable, appliquée à t_1 et t_2 respectivement, ne permet de les différencier. La notion de contexte observable permet d'exprimer formellement cette idée.

Définition 2 : Etant donnée une signature Σ, un *contexte* est un Σ-terme avec une et une seule variable.

Une *spécification observationnelle* est une spécification classique munie d'un ensemble de

contextes $\mathcal{C}_{Obs}$ (éventuellement infini) dont les éléments sont appelés des *contextes observables.*

Une Σ-algèbre A *satisfait observationnellement* une équation $t_1 = t_2$ relativement à l'ensemble de contextes observables $\mathcal{C}_{Obs}$ si et seulement si elle satisfait toutes les équations de la forme $C(t_1) = C(t_2)$ pour C parcourant $\mathcal{C}_{Obs}$. ($C(t)$ est une abréviation pour $C[x \leftarrow t]$ où x est la variable de C).

Une algèbre A satisfait une spécification observationnelle $SPEC$, si et seulement si elle satisfait observationnellement toutes les instances de tous les axiomes de $SPEC$ (via les tables de vérité des connecteurs ; pour simplifier, toutes les variables sont universellement quantifiées).

Par exemple, une approche fondée sur la déclaration d'un sous-ensemble S_{Obs} de sortes observables se contentera de définir $\mathcal{C}_{Obs}$ comme l'ensemble de tous les termes de sorte $s \in S_{Obs}$ avec une et une seule variable. L'article reproduit en chapitre 5 propose un affinement de cette approche. Pour notre exemple, puisque l'on ne veut pas que les résultats issus de *enum* soient observables, on se restreint tout simplement aux contextes de sorte observable ne contenant aucune occurrence de *enum*. Ceci revient à déclarer, en plus de S_{Obs}, un sous-ensemble d'opérations de la signature qui « autorise » l'observation ($\Sigma_{Obs} = \Sigma - \{enum\}$).

Définition 3 : Une spécification observationnelle (dans ce cadre) est une spécification habituelle munie de deux sous-ensembles S_{Obs} et Σ_{Obs} de S et Σ respectivement. L'ensemble de contextes observables correspondant $\mathcal{C}_{Obs}$ est par définition l'ensemble des Σ_{Obs}-termes de sorte $s \in S_{Obs}$ avec une et une seule variable.

On doit maintenant se demander pourquoi les deux derniers axiomes de SET (figure 4, page 24) sont correctement traités avec cette sémantique bien qu'ils contiennent l'opération *enum*. La raison en est simple : *enum* est proscrit des contextes observables, mais pas de leurs instances. Ainsi l'axiome

$$redundant(enum(E)) = false$$

donne lieu à tous les

$$C(redundant(enum(E))) = C(false)$$

où C est un contexte de sorte Nat ou $Bool$ dont la variable est de sorte $Bool$. Ici la situation est particulièrement simple : puisque $\mathcal{C}_{Obs}$ contient en particulier le contexte réduit à la variable booléenne x, cet axiome doit être satisfait au sens classique (i.e. avec l'égalité de la théorie des ensembles). Si l'on revient maintenant à l'algèbre A décrite plus haut qui réalise Set par des listes, on doit vérifier directement, sans passer par une quelconque algèbre B, qu'elle satisfait observationnellement tous les axiomes de SET. Prenons par exemple

$$insert(n, insert(m, E)) = insert(m, insert(n, E))$$

qui n'est clairement pas satisfait avec l'égalité ensembliste. On constate même que

$$enum(insert(1, insert(2, \emptyset))) \neq enum(insert(2, insert(1, \emptyset))$$

parce que par exemple

$$1 = car(enum(insert(1, insert(2, \emptyset)))) \neq car(enum(insert(2, insert(1, \emptyset))) = 2$$

mais le contexte $car(enum(E))$, bien que de sorte observable (Nat), n'est pas élément de $\mathcal{C}_{Obs}$. Compte tenu de la définition de Σ_{Obs}, pour tout contexte observable des ensembles, $C(X)$, $C(insert(n, insert(m, E)))$ et $C(insert(m, insert(n, E)))$ conduisent toujours à des résultats identiques.

Plus précisément : on remarque tout d'abord que l'on peut se restreindre aux contextes observables

ne contenant aucun sous-contexte observable. De tels contextes $C(X)$ sont nécessairement de la forme $C(X) = [t \in C'(X)]$ où C' ne contient que les opérations $\emptyset$ ou *insert*. Il en résulte que le résultat de $C(X)$ ne peut pas dépendre de l'ordre d'insertion…

On voit donc qu'avec une spécification de ce qui est observable beaucoup plus simple que dans le cas d'une sémantique avec équivalence de modèles, on arrive néanmoins aux résultats attendus. L'article fournit de plus divers résultats d'initialité, foncteurs adjoints, etc, que nous ne détaillerons pas ici[28].

Là où les choses sont moins simples, c'est lorsqu'on étudie le comportement du foncteur d'oubli sur des spécifications observationnelles (quelle que soit la sémantique choisie). Prenons deux spécifications observationnelles $SPEC_1$ et $SPEC_2$ telles que $SPEC_1$ soit une sous-spécification de $SPEC_2$ (c'est-à-dire $S_1 \subseteq S_2$, $\Sigma_1 \subseteq \Sigma_2$, $S_{Obs1} \subseteq S_{Obs2}$, etc.)[29]. Etant donnée une Σ_2-algèbre A, son oubli est la Σ_1-algèbre $U(A)$ obtenue en ne conservant que les A_s pour lesquels $s \in S_1$ et en ne considérant que les opérations de Σ_1. Une propriété qui semblerait naturelle est que pour toute formule φ sur la signature « pauvre » Σ_1, et pour toute algèbre A sur la signature « riche » Σ_2, on ait :

$$U(A) \models \varphi \Longleftrightarrow A \models \varphi$$

Cette propriété est appelée la *condition de satisfaction* et est un point central de la théorie des institutions. Il est malheureusement bien clair que dans un cadre observationnel l'implication $\Longrightarrow$ n'est pas assurée. Il suffit en effet de considérer une formule de la forme $a = b$ et une spécification $SPEC_1$ munie d'une observabilité suffisamment faible pour que a ne soit pas différentiable de b (par exemple aucune observation), mais une spécification $SPEC_2$ munie d'une observation plus complète permettant de les distinguer (par exemple en observant directement la sorte de a et b). Il pourra exister alors une algèbre A dans laquelle a n'est pas égal à b au sens de la théorie des ensembles, $U(A)$ satisfera néanmoins $a = b$ avec l'observation de $SPEC_1$, mais A ne le satisfera pas avec l'observation de $SPEC_2$.

La raison de ce phénomène est que l'observation autorisée est contextuelle, c'est-à-dire attachée à la spécification entière, indépendamment des axiomes, alors qu'elle influe fortement sur leur sémantique. Une solution serait de modifier la définition d'un « axiome », et de lui adjoindre l'observation de laquelle il dépend. Un axiome serait alors un couple (φ, Obs). Cette approche est actuellement étudiée dans la thèse de Teodor Knapik[30], elle présente toutefois le désavantage de compliquer les spécifications. En pratique, il est naturel d'adjoindre une fois pour toute une observabilité donnée à un module de spécification, et qu'elle soit partagée par tous les axiomes. Par conséquent, considérant une formule φ quelconque et une Σ_2-algèbre A, il est plus naturel de traiter « $A \models \varphi$ » relativement à l'observation de $SPEC_2$ que relativement à celle de $SPEC_1$. En pratique, ce que l'on veut vraiment, ce n'est pas *modifier* la définition de la sémantique ou de la syntaxe pour obtenir que la condition de satisfaction soit *toujours* vraie, c'est au contraire *se servir* de cette condition comme d'un *critère* pour considérer que la spécification $SPEC_2$ est acceptable au-dessus de $SPEC_1$ ou non. Autrement dit, $SPEC_1$ étant donnée avec son observabilité, on ne veut écrire que des enrichissements qui préservent cette observation. Ce critère pour les spécifications observationnelles est en fait similaire au critère de consistance hiérarchique pour les spécifications classiques. Il n'en reste pas moins que c'est seulement à la lumière de travaux étudiant sous quelles conditions la condition de satisfaction est valide que nous pourrons dégager de bonnes méthodes pour caractériser les enrichissements acceptables. Quelques résultats, assez

[28] Ils sont néanmoins « amusants » en ce sens que l'observabilité conduit en général plutôt à des résultats de « terminalité », je ne m'attendais pas au départ à obtenir aussi facilement des résultats d'initialité…

[29] Plus généralement on considère souvent des morphismes de signature qui autorisent par exemple le renommage ou la réunion de plusieurs sortes, mais ici nous préférons conserver des exemples très simples.

[30] que nous encadrons Michel Bidoit et moi.

techniques mais relativement encourageants, sont d'ores et déjà établis dans l'article « *Towards an adequat notion of observation* », publié[31] en collaboration avec Teodor Knapik et Michel Bidoit (non reproduit dans ce document).

Il ne faut pas croire[32] que l'implication inverse soit toujours vraie. Les spécifications observationnelles la mettent en défaut aussi. Il suffit en effet de considérer une formule φ de la forme $(a = b \Longrightarrow c = d)$ et une algèbre A pour laquelle $a{\neq}b$ et $c{\neq}d$, les sortes de a et c étant différentes. Supposons que $SPEC_1$ et $SPEC_2$ fournissent une observation totale sur la sorte de c et d. On a alors

$$U(A) \not\models (c = d) \quad \text{et} \quad A \not\models (c = d)$$

Supposons, exactement comme pour l'exemple précédent, que $U(A)$ satisfasse observationnellement $a = b$ (avec l'observation de $SPEC_1$) mais que A ne le satisfasse pas (par exemple parce que $SPEC_2$ ajoute une observation totale sur la sorte de a et b). Rappelons que selon la table de vérité de l'implication, « faux implique faux » est vrai. Il en résulte que

$$U(A) \not\models \varphi \quad \text{mais} \quad A \models \varphi$$

Ce qui prouve qu'aucune des implications de la condition de satisfaction n'est vraie dans le cadre des spécifications observationnelles.

Une autre idée que nous défendons en chapitre 5 est que tout ceci n'a aucune importance. Il s'agit de phénomènes liés à la structuration *modulaire* des spécifications, et il ne faut donc pas chercher à les traiter au travers d'une sémantique qui « met à plat » la spécification $SPEC_2$. Comme nous venons de le dire, il faut au contraire exploiter la structuration issue de l'inclusion de $SPEC_1$ dans $SPEC_2$ pour se restreindre implicitement aux modèles de $SPEC_2$ qui protègent ceux de $SPEC_1$. Autrement dit, *l'observabilité est indissociable de la modularité*. En fait, des contraintes modulaires implicites sont le meilleur moyen d'atteindre un style de spécification réellement abstrait.

Pour appuyer cette affirmation, considérons l'exemple de spécification décrit en figure 4. Il

---

5— Ensembles d'entiers avec choix

Spec « SET ».
S : $\{Set\}$
Σ : $\emptyset \;:\to Set$
$insert \;:\; Nat \times Set \to Set$
$\in \;:\; Nat \times Set \to Bool$
$choose \;:\; Set \to Nat$
Ax : $n \in \emptyset = false$
$n \in insert(n, E) = true$
$eq(n, m) = false \Longrightarrow n \in insert(m, E)) = n \in E$
$choose(\emptyset) = 0$
$E{\neq}\emptyset \quad\Longrightarrow\quad choose(E) \in E = true$
Observation : les sortes *Bool* et *Nat*.

---

est clair qu'aucune information à propos de *choose* autre que $choose(E) \in E$ n'est pertinente[33]. Pour obtenir un style de spécification réellement abstrait, il ne faut pas être obligé de caractériser

[31] A paraître dans les Proc. ESOP, Rennes, Février 1992.
[32] contrairement aux affirmations des défenseurs de la théorie des « specification logics ».
[33] Oublions le traitement d'exceptions pour $choose(\emptyset)$, ce n'est pas notre propos dans cette section.

l'élément retenu par *choose*.
Parmi les algèbres satisfaisant cette spécification, certaines ajoutent des « junks » aux entiers naturels. En effet, une algèbre A telle que *choose* associe à un ensemble $\{a, b\}$ une nouvelle valeur $\alpha_{a,b}$ de sorte Nat, vérifiant simplement $\alpha_{a,b} \in \{a, b\} = true$, satisfait les axiomes de SET.
La raison pour laquelle une telle algèbre n'est pas attendue par le « spécifieur » est qu'il avait supposé implicitement que l'ensemble des réalisations correctes des spécifications NAT et BOOL étaient fixé une fois pour toute. On voit donc que pour écrire des spécifications qui ne décrivent que le « quoi » et non le « comment », il n'est pas judicieux de compléter la spécification pour que les *axiomes* impliquent que tout terme de sorte Nat soit réductible à un terme de la forme $succ^i(0)$. Il faut au contraire que la sémantique sous-jacente offre ce service implicitement, afin de strictement réduire l'ensemble d'axiomes requis à l'expression d'informations pertinentes.

Remarquons que nous n'avons écrit ici aucun axiome sur la permutabilité de *insert* (ou sur son idempotence). Compte tenu de l'observabilité restreinte que nous prenons, ce sont des théorèmes observationnels de la spécification. Nous n'avions écrit ces axiomes en figure 4 que pour faciliter la description des algèbres au sens classique qui nous ont été utiles pour commenter la sémantique par équivalence observationnelle d'algèbres. Il est par contre plus important de remarquer que même pour la spécification de la figure 4, nous avions exploité sans le mentionner une contrainte hiérarchique implicite : les axiomes n'imposaient en fait nullement que les listes obtenues par *enum* ne soient pas des junks... Autrement dit, nous n'avons spécifié le résultat de *enum* qu'au travers des observations *redundant* et *occurs* sur les listes. C'est donc là encore la combinaison d'une sémantique observationnelle et de contraintes hiérarchiques qui permet de ne spécifier que les informations utiles.

Exactement comme pour l'implémentation abstraite ou pour l'étape d'oracle du test de logiciel, le point le plus difficile lié aux spécifications observationnelles est la preuve de théorèmes. Alors qu'il est en pratique relativement aisé de prouver que deux valeurs sont observationnellement distinctes (il suffit d'exhiber un contexte qui les distingue), il est très difficile de prouver qu'elles sont observationnellement égales. En effet, cette propriété met en jeu un ensemble infini de contextes observables. Le seul travail existant[34] sur ce sujet est celui de R. Hennicker qui utilise pour ce faire une méthode d'induction sur les contextes [Hen90]. Cette dernière n'est applicable à l'heure actuelle que pour une observation de sortes. Puisque notre définition met en jeu des contextes construits sur un sous-ensemble connu de la signature, il est probable que les travaux de R. Hennicker soient applicables à notre approche. Ceci n'a toutefois pas encore été étudié, et contrairement à la sémantique « classique » nous ne savons pas encore quel sera l'impact des contraintes sémantiques modulaires sur ces techniques de preuve de théorèmes avec observabilité.

5 Les algèbres étiquetées

> Aphorisme : *même si deux termes possèdent des valeurs égales, il se peut que, dans un axiome, l'un soit une occurrence acceptable d'une variable, mais pas l'autre.*

Les détails techniques relatifs à cette section sont consignés dans l'article intitulé « *Label algebras and exception handling* »reproduit en chapitre 6.

Une propriété un peu déroutante des sémantiques algébriques de l'observabilité fondées sur la notion de contextes observables est que même si deux termes possèdent la même valeur, il est possible que l'un soit observable mais pas l'autre. Revenons sur la spécification SET donnée en

[34]à ma connaissance.

figure 4. Nous nous sommes convaincus que tout résultat fourni par *enum* ne devait être observé que de manière restreinte. Plus précisément, les contextes observables de $\mathcal{C}_{Obs}$ ne doivent pas contenir l'opération *enum*. En conséquence, nous avons réussi à prouver la satisfaction observationnelle de l'axiome sur la permutabilité de *insert* bien que

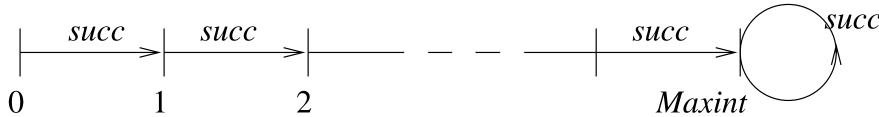
$$1 = car(enum(insert(1, insert(2, \emptyset)))) \neq car(enum(insert(2, insert(1, \emptyset)))) = 2$$

car les termes de la forme $car(enum(E))$ ne sont pas observables. Pourtant, puisque les termes $succ(0)$ et $succ(succ(0))$ sont de sorte Nat et ne contiennent pas *enum*, ils sont observables. Donc les termes $succ(0)$ et $car(enum(insert(1, insert(2, \emptyset))))$ possèdent des valeurs égales (elles sont même égales selon l'égalité ensembliste), mais le premier est observable et non le second. Ceci n'induit cependant aucune inconsistance.

Une autre occurrence de ce phénomène apparaît pour les sémantiques algébriques avec traitement d'exceptions. De même que dans les langages de programmation avec traitement d'exceptions on distingue deux parties de texte avec des sémantiques différentes (le déroulement « normal » de l'algorithme et les « handlers » d'exceptions), on distingue pour les spécifications avec traitement d'exceptions deux types d'axiomes avec des sémantiques différentes : les « *Ok*-axiomes » (qui traitent les cas « normaux ») et les axiomes dédiés au traitement d'exceptions. C'est par exemple le cas de la théorie des exception-algèbres décrite dans l'article « *Abstract data types with exception handling : an initial approach based on a distinction between exceptions and errors* »[35], écrit en collaboration avec Michel Bidoit et Christine Choppy (non reproduit dans ce document). Supposons par exemple que l'on veuille spécifier une structure d'entiers naturels bornés. Lorsqu'aucune exception n'est levée, la propriété suivante doit être vérifiée :

$$pred(succ(n)) = n$$

elle est donc placée parmi les *Ok*-axiomes. Soit $Maxint$ la borne supérieure. Le terme $succ(Maxint)$ est alors un terme exceptionnel. On peut éventuellement lui appliquer un traitement d'exceptions. On peut par exemple vouloir le *récupérer* sur $Maxint$ lui même.



Ceci est alors traduit en plaçant l'axiome suivant parmi les axiomes de traitement d'exceptions :

$$succ^{Maxint+1}(0) = succ^{Maxint}(0)$$

Le point important est alors de remarquer que, bien que les termes $succ^{Maxint+1}(0)$ et $succ^{Maxint}(0)$ possèdent des valeurs égales, le premier est un terme exceptionnel alors que le second est un terme « normal » (appelé un *Ok*-terme). Si l'on n'autorise pas cette différence fondamentale entre deux *termes* de même *valeur*, alors soit ces deux termes sont des *Ok*-termes, soit aucun des deux n'est un *Ok*-terme. Dans le premier cas, le terme $succ(succ^{Maxint}(0))$ [36] serait un *Ok*-terme, et par conséquent, puisque l'opération *pred* ne lève aucune exception sur la valeur $Maxint$, le terme $pred(succ(succ^{Maxint}(0)))$ serait lui aussi un *Ok*-terme. Il en résulterait que l'occurrence suivante de notre *Ok*-axiome s'applique :

$$pred(succ(succ^{Maxint}(0))) = succ^{Maxint}(0)$$

Mais par ailleurs $succ^{Maxint+1}(0) = succ^{Maxint}(0)$, donc il viendrait :

$$pred(succ(succ^{Maxint}(0))) = pred(succ^{Maxint}(0)) = succ^{Maxint-1}(0)$$

[35] Paru dans le journal T.C.S., Vol.46, No.1, p.13-45, Novembre 1986.

[36] qui est *syntaxiquement*, c'est-à-dire en tant que terme, égal à $succ^{Maxint+1}(0)$.

si bien que nous obtiendrions l'inconsistance suivante

$$succ^{Maxint}(0) = succ^{Maxint-1}(0)$$

qui, se propageant de la même façon, conduirait à ce que tous les entiers soient égaux à 0. Dans le second cas (c'est-à-dire si aucun des deux termes n'est un *Ok*-terme), $succ^{Maxint}(0)$ n'étant pas un *Ok*-terme, $pred(succ^{Maxint}(0))$ ne serait a fortiori pas un *Ok*-terme[37]. Donc l'occurrence

$$pred(succ^{Maxint}(0)) = succ^{Maxint-1}(0)$$

ne s'appliquerait pas et l'opération *pred* serait un nouveau constructeur des entiers naturels (elle créerait des « junks »). Il est donc bien clair que le traitement d'exceptions doit porter sur des *termes exceptionnels* et non sur leur *valeur* qui n'est pas nécessairement erronée. Même si un terme exceptionnel est récupéré sur une valeur *Ok*, il n'en reste pas moins exceptionnel.

Par ailleurs, il est bien clair que, même si les termes $succ^{Maxint+1}(0)$ et $pred(0)$ sont tous deux exceptionnels, ils ne répondent pas au même type d'exception et ne doivent pas nécessairement être traités de la même façon (par exemple on peut vouloir n'en récupérer qu'un des deux). Pour faciliter le style de spécification, il est nécessaire de pouvoir *nommer* le type d'exception auquel ils répondent respectivement. Pour écrire des spécifications claires et précises, il est utile de pouvoir spécifier par exemple des axiomes tels que :

$$succ^{Maxint+1}(0) \;\epsilon\; TooLarge$$

$$pred(0) \;\epsilon\; Negative$$

$$n < m = true \Longrightarrow (n - m) \;\epsilon\; Negative$$

Bien évidemment, bien que $succ^{Maxint+1}(0)$ soit récupéré sur $succ^{Maxint}(0)$, on ne doit aucunement en déduire que $succ^{Maxint}(0)$ lève le nom d'exception *TooLarge*. Il est par conséquent nécessaire de dégager différentes classes de termes qui ne sont pas nécessairement compatibles avec les valeurs. Nous appellerons « étiquettes » les « noms de classes de termes ». Alors que les *sortes* portent sur les valeurs, les *étiquettes* portent sur les termes.

C'est pour cette raison que nous avons développé, Pascale Le Gall et moi, la théorie des *algèbres étiquetées*. Le chapitre 6 développe cette théorie et en démontre l'utilité pour le traitement d'exceptions (divers travaux d'extension de ce formalisme sont prévus et/ou en cours). Au niveau de la syntaxe, puisque les étiquettes ne sont ni des opérations, ni des prédicats, ni des sortes qui portent sur des valeurs, on ajoute dans la signature un ensemble d'étiquettes. La syntaxe des axiomes est alors étendue pour prendre en compte des atomes d'étiquetage de termes.

Définition 4 : Une *signature étiquetée* est un triplet $\Sigma L = (S, \Sigma, L)$ où (S, Σ) est une signature classique et L est un ensemble d'*étiquettes*[38].

Etant donnée une signature étiquetée, un *atome équationnel* est une équation au sens habituel $(t_1 = t_2)$ entre deux Σ-termes (avec variables) de même sorte, et un *atome d'étiquetage* est une expression de la forme $(t \;\epsilon\; l)$ où t est un Σ-terme avec variables et l est une étiquette de L. Le symbole « ϵ » se lit « est étiqueté par ». Une formule bien formée, ou *axiome*, est alors construite à partir des atomes et de connecteurs logiques quelconques.

Un axiome est dit *conditionnel positif* s'il est de la forme

$$(a_1 \;\wedge \cdots \wedge\; a_n) \Longrightarrow a_{n+1}$$

où les a_i sont des atomes.

Une *spécification étiquetée* est formée d'une signature étiquetée et d'un ensemble de formules bien formées sur cette signature.

[37] Cette propriété est appelée la *propagation d'exceptions*.

[38] L comme « labels ».

Toutes les variables sont donc implicitement universellement quantifiées dans les axio-mes. Nous n'avons pas encore étudié l'extension autorisant des quantificateurs existentiels (elle ne devrait pas poser de problème majeur).

Il faut ensuite définir ce qu'est une algèbre sur une signature étiquetée. Puisqu'une signature étiquetée contient en particulier une signature classique, une algèbre étiquetée sera une Σ-algèbre classique munie de renseignements additionnels traduisant la sémantique des étiquettes. Comme on l'a vu, pour donner une sémantique à chaque étiquette, il faut lui associer non pas un sous-ensemble des valeurs de l'algèbre, mais un ensemble de termes. Si nous nous limitions à un ensemble de termes fermés sur la signature, nous ne pourrions clairement pas traiter de manière satisfaisante des algèbres non finiment engendrées. En effet, aucune valeur non atteignable par les opérations de la signature ne pourrait contenir (dans la classe d'équivalence de termes associée) un terme étiqueté. La question qui se posait alors était donc de fournir une extension de la notion de terme pour pouvoir étiqueter des objets non représentables par des Σ-termes fermés. Cette question est en fait une question standard en algèbre, et a été résolue depuis longtemps en considérant des termes sur la signature dont les feuilles peuvent être soit des constantes de la signature, soit des valeurs de l'algèbre. Il s'avère que, comme dans le cadre des « exception-algèbres », nous pouvons appliquer cette technique avec succès.

Rappel 5 : Soient (S, Σ) une signature et V un ensemble de variables « typées » (c'est-à-dire que V est une union disjointe d'ensembles de variables V_s pour $s \in S$). On notera $T_\Sigma(V)$ l'algèbre libre des termes avec variables dans V. Puisque V n'est pas nécessairement fini ni même dénombrable dans cette définition, on peut considérer, étant donnée une Σ-algèbre A, l'algèbre libre des termes avec « variables » dans A : $T_\Sigma(A)$.

Remarquons que A est inclus de manière canonique dans $T_\Sigma(A)$ (c'en est un sous-ensemble de constantes). On note « *eval* » le Σ-morphisme canonique de $T_\Sigma(A)$ dans A qui prolonge l'identité sur A.

Par exemple, si $\Sigma = \{0, succ\}$ est la signature des entiers naturels, et $A = \mathbb{Z}$, alors $T_\Sigma(\mathbb{Z})$ contient en particulier les termes -2, $succ(-3)$, $succ(succ(-4))$, etc, qui sont *distincts* bien que leurs valeurs, après évaluation dans $\mathbb{Z}$, soient égales.

On peut dès lors caractériser la sémantique d'une étiquette par un sous-ensemble de $T_\Sigma(A)$, ce qui permet de prendre en compte les algèbres non finiment engendrées.

Définition 6 : Etant donnée une signature étiquetée $\Sigma L = (S, \Sigma, L)$, une *algèbre étiquetée* est un couple $\mathcal{A} = (A, \{l_A\}_{l \in L})$ où A est une Σ-algèbre classique et $\{l_A\}_{l \in L}$ est une famille indexée par L de sous-ensembles l_A de $T_\Sigma(A)$.

Pour $l \in L$, les éléments de l_A sont donc bien des termes et non des valeurs de A.

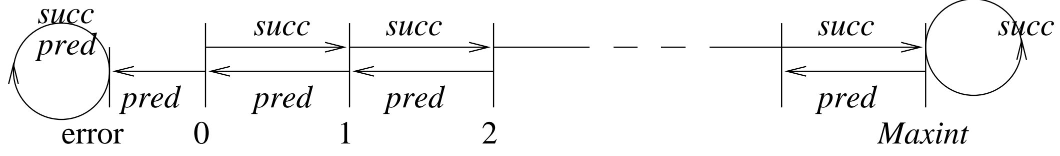
Il ne reste plus qu'à définir la notion de satisfaction d'un axiome par une algèbre. Le point crucial ici est que nous considérons des substitutions à valeurs dans les *termes* de $T_\Sigma(A)$, et non pas dans les *valeurs* de A. Ceci est majeur pour limiter aux termes seulement les étiquetages imposés par un atome d'étiquetage. Par contre, il est clair que les atomes équationnels doivent se référer aux valeurs, c'est pourquoi nous utilisons le morphisme d'évaluation dans ce cas.

Définition 7 : Soit $\mathcal{A} = (A, \{l_A\}_{l \in L})$ une ΣL-algèbre,

- étant donnés $u \in T_\Sigma(A)$ et $l \in L$, « $\mathcal{A} \models (u \,\epsilon\, l)$ »signifie $u \in l_A$
- étant donnés $v \in T_\Sigma(A)$ et $w \in T_\Sigma(A)$, « $\mathcal{A} \models (v = w)$ »signifie $eval(v) = eval(w)$ (dans A).
- étant donné un axiome φ, « $\mathcal{A} \models \varphi$ » signifie que pour toute substitution σ des variables de φ *à valeur dans* $T_\Sigma(A)$, $\mathcal{A} \models \sigma(\varphi)$ en suivant les tables de vérité des connecteurs et la satisfaction atomique définie par les deux points précédents.

Revenons à notre exemple des entiers naturels bornés par *Maxint*, et considérons l'algèbre $\mathcal{A} = (A, \{l_A\}_{l \in L})$ définie par :

– A est la Σ-algèbre décrite par : *pred* est l'opération inverse de *succ* « dans la ligne droite » et $pred(0)$ retourne une constante erronée sur laquelle *succ* et *pred* seront ensuite idempotentes ; c'est-à-dire le dessin suivant



– $TooLarge_A$ est l'ensemble des termes de la forme $succ(t)$ où t est un terme de $T_\Sigma(A)$ tel que $eval(t) = Maxint$. C'est par exemple le cas des termes : $succ^{Maxint+1}(0)$, $succ^{Maxint}(1)$, $succ(succ(pred(Maxint)))$, etc, mais ce n'est pas le cas de $succ^{Maxint}(0)$ bien qu'il possède la même valeur.

– $Negative_A$ est l'ensemble des termes t de $T_\Sigma(A)$ tels que $eval(t) = \text{error}$. C'est par exemple le cas de $pred(0)$, error, $succ(\text{error})$, etc.

– Enfin Ok_A est l'ensemble de termes construit récursivement comme suit : il contient tous les termes constants sauf « error »[39], tous les termes de la forme $succ(t)$ tels que $t \in Ok_A$ et $eval(t) \neq Maxint$, et tous les termes de la forme $pred(t)$ tels que $t \in Ok_A$ et $eval(t) \neq 0$.

L'axiome « $pred(succ(n)) = n$ » étant un *Ok*-axiome, il ne doit s'appliquer qu'à des *Ok*-termes et se traduit donc dans notre approche par :

$$pred(succ(n)) \mathrel{\epsilon} Ok \quad \wedge \quad n \mathrel{\epsilon} Ok \implies pred(succ(n)) = n$$

On voit dès lors que pour l'algèbre $\mathcal{A}$ décrite précédemment, l'instance $n = Maxint$ ne satisfait pas la prémisse de l'axiome car $pred(succ(Maxint))$ n'est pas étiqueté par *Ok*. Ceci résout donc l'inconsistance décrite plus haut. Il n'est reste pas moins que la valeur de ce terme est récupérée puisque $eval(pred(succ(Maxint))) = Maxint - 1$.

Nous avons décrit ici « à la main » l'algèbre $\mathcal{A}$ et avons de même complété la prémisse de l'*Ok*-axiome. L'article du chapitre 6 montre comment une « exception-spécification » peut être automatiquement traduite en une spécification étiquetée pour éviter cet inconvénient. Bien d'autres résultats y sont consignés, en particulier des résultats d'initialité pour les spécifications positives conditionnelles qui étendent entièrement ceux bien connus dans le cas classique.

Un résultat par contre plus surprenant est que les algèbres étiquetées ne valident pas la « condition de satisfaction » déjà mentionnée dans la section 4 (page 27). Rappelons sa définition. On se donne deux signatures étiquetées ΣL_1 et ΣL_2 telles que $\Sigma L_1 \subseteq \Sigma L_2$ (membre à membre), une ΣL_1-formule φ et une ΣL_2-algèbre $\mathcal{A} = (A, \{l_A\}_{l \in L_2})$; cette condition s'exprime par :

$$U(\mathcal{A}) \models \varphi \iff \mathcal{A} \models \varphi$$

L'oubli $U(\mathcal{A}) = \mathcal{B} = (B, \{l_B\}_{l \in L_1})$ étant défini de manière évidente par :

– $B = U(A)$ au sens des Σ_1-algèbres classiques

– $l_B = l_A \cap T_{\Sigma_1}(B)$ pour tout $l \in L_1$.

Il n'est pas très difficile de prouver que l'implication de droite à gauche ($\Longrightarrow$) de la condition de satisfaction est toujours vraie ; par contre l'implication inverse est fausse, comme le montre le contre-exemple suivant, mis en évidence par Pascale Le Gall :

Exemple 8 : Considérons la signature étiquetée ΣL_1 définie par

$S_1 = \{ \, LaSorte \, \}$
$\Sigma_1 = \{ \, cste1 \; :\to LaSorte \, \}$
$L_1 = \{ \, LeLabel \, \}$.

[39]c'est-à-dire les valeurs 0 à *Maxint* de A et la constante 0 de la signature (qui, en tant que terme, n'est pas égale à la valeur 0 de A...)

Considérons maintenant ΣL_2 définie par

$S_2 = S_1$
$\Sigma_2 = \{ cste1,\ cste2 \ :\rightarrow LaSorte \}$
$L_2 = L_1$

et considérons enfin la Σ_2-algèbre $\mathcal{A}$ définie par

$A = \{ a{=}cste1{=}cste2 \}$ (i.e. la Σ_2-algèbre sous-jacente est un singleton)
$LeLabel_A = \{ a,\ cste1 \}$ [40].

$U(\mathcal{A})$ est alors l'algèbre étiquetée telle que

$U(A) = \{ a = cste1 \}$ et $LeLabel_{U(A)} = \{ a,\ cste1 \}$.

Ainsi $LeLabel_{U(A)} = T_{\Sigma_1}(U(A))$. $U(\mathcal{A})$ satisfait donc l'axiome « $x \in LeLabel$ ». Pourtant $\mathcal{A}$ ne satisfait pas cet axiome car $cste2 \notin LeLabel_A$.

Ce contre-résultat n'a, jusqu'à maintenant, eu aucune conséquence fâcheuse sur les applications des algèbres étiquetées que nous avons étudiées. Par contre, de nombreuses extensions de cette théorie sont envisageables, pour ne pas dire nécessaires. Mentionnons en particulier l'étude d'un ensemble de règles d'inférence correct et complet pour ce formalisme. Les travaux développés pour l'Equational Type Logic par V. Manca, A. Salibra et G. Scollo [MSS89] pourront être d'une grande utilité, d'autant plus que nous aimerions aussi étudier une théorie des algèbres étiquetées d'ordre supérieur. Nous avons déjà constaté qu'il était aisé de spécifier des ensembles de termes observables, mais il reste à étudier en détail comment spécifier diverses sémantiques observationnelles dans le cadre des algèbres étiquetées. Pour ce faire, il faudra probablement considérer une « étiquette binaire » sur les termes qui traduise l'égalité observationnelle. C'est entre autres pour cette raison que nous prévoyons une extension des algèbres étiquetées à des étiquettes d'arité quelconque. En effet, les étiquettes sont pour le moment des sortes de « prédicats » unaires sur les termes. A priori, il ne devrait pas être difficile de les rendre n-aires. Dans le chapitre 6, nous définissons une « contrainte d'évaluation partielle ». Pour le moment cette contrainte ne peut pas être directement spécifiée par des axiomes étiquetés. Ceci serait rendu possible par la manipulation du morphisme d'évaluation au sein même des axiomes. Il reste donc à étudier quels sont les résultats qui restent valables avec cette extension. Bien d'autres extensions sont envisageables, incluant l'introduction de quantificateurs existentiels. L'atout majeur des algèbres étiquetées est la différenciation systématique des *termes* et des *valeurs*, qui permet, comme nous l'avons vu sur un exemple, de restreindre la portée d'un axiome de manière très fine. On peut en effet non seulement restreindre cette portée à des valeurs particulières d'une sorte donnée, mais on peut aussi la restreindre à des motifs[41] de termes très particuliers. Ceci est la clef qui nous a permis par exemple de définir une sémantique du traitement d'exceptions sans lever les inconsistances qui semblaient jusqu'alors inévitables. Il semble clair qu'il existe de nombreuses autres sémantiques particulières (telle l'observabilité) qui pourront bénéficier de cet « outil contre les inconsistances ».

6 Les « métathéories »

Aphorisme[42] : *Les bons foncteurs... sont ceux qui préservent la philosophie !*

Comme l'attestent les sections qui précèdent, il existe de nombreuses théories des types abstraits algébriques. Toutes sont à divers égards des extensions de la théorie « classique » qui fut proposée par le groupe ADJ en 1978 [GTW78]. Dans la mesure où certaines d'entre elles traitent

[40] Rappelons que $T_{\Sigma_2}(A) = \{a, cste1, cste2\}$
[41] i.e. patterns...
[42] L'auteur de cet aphorisme est Stéphane Kaplan.

des questions apparemment indépendantes, on peut se demander s'il n'existerait pas un moyen de bénéficier de plusieurs extensions à la fois, sans refaire une nouvelle théorie qui les généraliserait ponctuellement. La théorie des catégories a fourni quelques avancées dans ce domaine, mais par sa généralité même, elle ne permet pas de dégager aisément des résultats spécifiques aux types abstraits algébriques. C'est un peu l'esprit dans lequel la théorie des institutions, que nous avons déjà mentionnée occasionnellement dans les sections précédentes, a été conçue par J.A. Goguen et R.M. Burstall en 1984 [GB84]. L'idée est de dégager les points communs à toutes ces théories, de les définir formellement, et de fournir ensuite des « métarésultats » applicables à toute théorie qui répond à la définition d'institution. Cette « montée d'un niveau » dans l'abstraction est clairement fructueuse et permet au minimum de donner des outils formels ou des critères non empiriques de comparaison entre les théories. Un sujet sur lequel la théorie des institutions a permis de grandes avancées indépendantes d'une sémantique particulière est par exemple l'étude de la modularité des spécifications. Mentionnons en particulier la sémantique par classes stratifiées de modèles proposée par Michel Bidoit en 1989 [Bid89]. On assiste actuellement à une sorte d'explosion de variantes, d'extensions ou de solutions de rechange à la « métathéorie » des institutions. Le but de cette section n'est pas d'entrer dans les diverses polémiques que cela soulève, mais de soumettre quelques « besoins d'un utilisateur naïf ».

Nous allons commencer par discuter (relativement indépendamment de l'approche choisie) des rapports entre les algèbres retenues par une sémantique donnée et les réalisations correctes d'une spécification. Les résultats auxquels nous nous réfèrerons alors (en section 6.1) sont relativement anciens, issus de l'article « *Good functors are those preserving philosophy* » (chapitre 7). Ils plaident en faveur d'une *sémantique modulaire* des types abstraits algébriques. Nous ferons ensuite quelques commentaires sur la sémantique modulaire par classes stratifiés de modèles (section 6.2) et enfin sur quelques aspects relatifs aux institutions ou métathéories similaires (section 6.3).

6.1 Sémantiques initiales et sémantiques « loose »

Aphorisme : *La théorie des catégories pour traiter une sémantique non modulaire des types abstraits algébriques, c'est un peu comme un tracteur pour labourer son jardin.*

La première sémantique des spécifications algébriques, proposée par ADJ, était une *sémantique initiale* [GTW78]. Il est maintenant largement admis que cette sémantique ne conduit pas à un style de spécification très satisfaisant, et c'est pourquoi de nombreuses extensions ont été proposées. Etant donnée une spécification $SPEC$, une sémantique initiale privilégie un unique modèle dans $Alg(SPEC)$: le modèle initial. Il en résulte que l'intuition fournie par cette sémantique ne peut pas être que les modèles retenus traduisent toutes les réalisations correctes d'une spécification, puisqu'il existe généralement plusieurs réalisations correctes non isomorphes d'une même spécification. Le but de cette sémantique est plutôt de fournir *le* modèle abstrait « auquel le spécifieur pense ». Il en résulte aussi qu'il n'est pas possible de prendre en compte dans cette approche simple des spécifications incomplètes. C'est le cas de la spécification des ensembles avec *choose*, où l'on ne souhaite pas préciser l'élément choisi dans un ensemble : plusieurs réalisations non isomorphes de l'opération *choose* sont acceptables. Dans l'algèbre initiale, *choose(E)* est un « junk », et il faudrait passer à des quotients de cette algèbre pour obtenir les modèles souhaités. Remarquons de plus que, par définition, le modèle initial (s'il existe) est celui qui différencie le plus de valeurs distinctes possibles. Par conséquent, pour rendre deux valeurs égales, il est nécessaire d'écrire plus d'axiomes que dans une sémantique « loose »ou observationnelle par exemple. Dans le cas de la spécification des ensembles en particulier, il faut explicitement fournir

les axiomes suivants pour obtenir l'algèbre initiale souhaitée :

$$insert(n, insert(m, E)) = insert(m, insert(n, E))$$

$$insert(n, insert(n, E)) = insert(n, E)$$

sous peine d'obtenir une spécification des listes. Souvent, ceci nuit à un style de spécification réellement abstrait et concis.

On pourrait croire cependant que l'on peut retrouver l'idée selon laquelle la classe des modèles d'une spécification représente toutes les réalisations acceptables en abandonnant simplement la restriction au modèle initial et en se contentant de considérer *toutes* les algèbres validant la spécification. Le fait qu'en abandonnant les deux axiomes sur *insert*, mentionnés plus haut, nous retrouvions la réalisation par les listes peut nous conforter dans cette idée. Malheureusement il n'en est rien, et à plus d'un égard. Remarquons tout d'abord que, considérant tous les modèles, on considère toujours en particulier le modèle initial. Donc l'argument sur les spécifications incomplètes telles que SET avec *choose* est toujours valable puisque, dans ce cas, l'algèbre initiale ne doit pas être considérée comme une réalisation correcte (à cause des junks engendrés par $choose(E)$). Remarquons aussi qu'il n'est pas toujours possible d'obtenir toutes les réalisations correctes en supprimant certains axiomes d'une approche initiale. Par exemple, pour la spécification bien connue des piles (figure 6.1), on serait tenté de supprimer les deux premiers axiomes.

6— L'exemple inévitable. . .

Spec « STACK ».

S : $\{Stack\}$

Σ : $empty$ $:\rightarrow Stack$

$push$: $Nat \times Stack \rightarrow Stack$

pop : $Stack \rightarrow Stack$

top : $Stack \rightarrow Nat$

Ax : $pop(empty) = empty$

$pop(push(e, P)) = P$

$top(empty) = 0$

$top(push(e, P)) = e$

En effet, considérons la réalisation classique des piles par des couples de tableaux et entiers : la pile vide est réalisée par la création d'un tableau initial et l'entier 0, *push* ajoute l'élément dans le tableau à l'index pointé par l'entier puis incrémente cet entier, et *pop* se contente de décrémenter l'entier. Cette réalisation ne satisfait pas le second axiome de la figure 6.1, car après avoir effectué *push* puis *pop*, le tableau a été modifié. Malheureusement, il n'est pas possible de supprimer les deux premiers axiomes : ils ne pourraient être remplacés que par une infinité d'axiomes de la forme $top(pop(pop(\cdots(push(e_1, push(e_2, \cdots(P)\cdots) = e$. Il n'est donc pas vrai en général qu'il suffise de réduire le nombre d'axiomes pour capturer toutes les réalisations correctes. C'est en fait uniquement par chance que ceci était possible avec les ensembles.

Une dernière remarque, beaucoup plus grave, est que les sémantiques initiales ne sont pas bien adaptées pour définir une sémantique modulaire des spécifications. En effet, considérons deux spécifications $SPEC_1$ et $SPEC_2$ telles que $SPEC_1 \subseteq SPEC_2$ (membre à membre, c'est-à-dire $S_1 \subseteq S_2$, $\Sigma_1 \subseteq \Sigma_2$, etc.). Ce dont on a réellement besoin, c'est de donner directement une sémantique à part entière au *morceau de spécification* $\Delta SPEC = (S_2 - S_1\,, \Sigma_2 - \Sigma_1\,, \cdots)$. Sachant que les foncteurs adjoints à gauche préservent les modèles initiaux, on peut penser qu'il suffise de considérer le *foncteur de synthèse* F_Δ, adjoint à gauche au foncteur d'oubli U. Cette

approche est malheureusement non viable, car ce foncteur F_Δ ne dépend pas uniquement de $\Delta SPEC$ il dépend aussi de $SPEC_1$. Nous devrions donc écrire $F_{SPEC_1}^{SPEC_2}$. Plus précisément, même si $SPEC_1$ et $SPEC'_1$ possèdent la même sémantique I (c'est-à-dire le même modèle initial dans cette approche), $F_{SPEC_1}^{SPEC_2}(I)$ n'est pas nécessairement isomorphe à $F_{SPEC'_1}^{SPEC_2}(I)$. Un exemple très simple de ce phénomène est décrit en chapitre 7. Ceci contredit bien évidemment tous les principes élémentaires de modularité. Pour donner une véritable sémantique modulaire des spécifications algébriques, il faut une sémantique fondée sur les *modèles* de $SPEC_1$, et non pas sur sa forme syntaxique particulière.

On voit donc qu'il faut « se méfier » des outils généraux fournis par la théorie des catégories. Même si un résultat général de la théorie des catégories (tel que la préservation des modèles initiaux par un foncteur) semble assurer une bonne sémantique, il n'en reste pas moins que le niveau d'abstraction fourni par cette théorie peut masquer des phénomènes indésirables. C'est en ce sens que nous justifions l'aphorisme énoncé au début de cette sous-section. Plus sérieusement, on peut se demander si les résultats très puissants fournis par la théorie des catégories sont réellement utilisés dans le cadre des types abstraits algébriques. En effet, les catégories que nous considérons sont particulièrement pauvres. Par exemple, on se limite souvent à des catégories d'algèbres finiment engendrées sur une signature ; or il existe *au plus un morphisme* entre deux algèbres finiment engendrées. Il semble par conséquent que la théorie des catégories fournisse essentiellement un cadre simple et un langage unifié pour traiter des théories sémantiques des types abstraits algébriques. Elle est par là même indispensable, mais il faut toujours rester prudent quant aux jugements erronés que cette généralité de langage facilite.

Là où les apports unificateurs de la théorie des catégories sont cependant appréciables, c'est lorsque l'on considère une sémantique réellement modulaire, avec des contraintes sémantiques implicites. Ces dernières sont en effet le seul moyen de traiter simplement des exemples tels que *choose* ou, comme nous l'avons déjà motivé en section 4, d'aborder l'observabilité avec un style de spécification réellement abstrait. L'article du chapitre 7 étudie quelques propriétés d'une contrainte sémantique implicite traduisant la protection par $\Delta SPEC$ des algèbres validant $SPEC_1$. De bien meilleures sémantiques de la modularité ont été définies depuis sa parution.

6.2 Sémantiques « loose » avec contraintes

Pour obtenir des spécifications qui expriment uniquement ce que l'on attend du logiciel, et non pas un quelconque moyen de mise en œuvre, nous avons déjà remarqué qu'il fallait assurer *implicitement* par la sémantique modulaire sous-jacente la protection des structures utilisées ($SPEC_1$). C'est par exemple le meilleur moyen de contraindre *choose* à retourner un élément préexistant sans surspécifier cette opération. Si l'on note $\mathcal{M}_1$ la classe des algèbres retenues par la sémantique de $SPEC_1$, cette contrainte s'exprime simplement par le fait que l'on ne retiendra que des Σ_2-algèbres A telles que $U(A) \in \mathcal{M}_1$.

Par ailleurs, le style de spécification est considérablement amélioré si l'on peut déclarer un *ensemble de générateurs*. La sémantique doit alors assurer automatiquement que, pour toute algèbre retenue par elle, toute valeur de l'algèbre est atteignable par un terme construit sur ces générateurs. Prenons un exemple pour appuyer cette affirmation. Considérons la propriété suivante sur les entiers naturels (où $b \neq 0$) :

$$0 \leq a - (a\ div\ b) \times b < b$$

qui n'est autre que la définition abstraite de la division euclidienne (elle signifie que le reste est compris entre 0 et le diviseur). Supposons que nous traduisions directement cette propriété dans

une spécification algébrique. Nous écririons :

$$b\neq 0 \Longrightarrow 0 \leq a - (a\ div\ b) \times b = true$$

$$b\neq 0 \Longrightarrow b \leq a - (a\ div\ b) \times b = false$$

Cependant, ceci ne suffit pas à caractériser le résultat de la division de deux entiers naturels. Il existe des algèbres pour lesquelles $(a\ div\ b)$ est un « junk » et $(a - (a\ div\ b) \times b)$ est aussi un junk, pourvu que les termes $(0 \leq a - (a\ div\ b) \times b)$ et $(b \leq a - (a\ div\ b) \times b)$, par contre, aient pour valeur *true* et $false$ respectivement. En fait, la raison pour laquelle cette spécification de *div* est suffisante est que parmi les entiers naturels engendrés par 0 et *succ*, il existe un unique entier $(a\ div\ b)$ qui satisfasse cette propriété. Il est bien clair que ces deux axiomes expriment complètement le « quoi », mais non le « comment », et qu'ils sont nettement préférables aux axiomes suivants

$$b\neq 0\ \wedge\ b \leq a = false \Longrightarrow a\ div\ b = 0$$

$$b\neq 0\ \wedge\ b \leq a = true \Longrightarrow a\ div\ b = succ((a - b)\ div\ b)$$

qui n'autorisent aucun junk mais expriment déjà un choix de mise en œuvre. Cet exemple démontre donc clairement l'utilité d'une restriction *implicite* aux algèbres finiment engendrées sur les générateurs.

Enfin, nous avons dit qu'il fallait donner aussi une sémantique à des « morceaux de spécification » $\Delta SPEC$. Le choix de la « sémantique par classes stratifiées de modèles » ([Bid89]) est de retenir la classe des *foncteurs* F allant de $\mathcal{M}_1$ dans la catégorie des $SPEC_2$-algèbres finiment engendrées sur les générateurs, vérifiant la contrainte de protection des algèbres de $\mathcal{M}_1$, qui s'exprime par :

$$(\forall A \in \mathcal{M}_1)(\ U(F(A)) = A\ )$$

La classe $\mathcal{M}_2$ est alors par définition la réunion des images des foncteurs retenus. C'est donc une sous-catégorie pleine de $Alg(SPEC_2)$. On peut alors facilement caractériser la sémantique d'une spécification modulaire en itérant ce procédé le long de la hiérarchie de la spécification. Pour ce faire, il faut encore définir la sémantique dans le « cas de base », c'est-à-dire pour les spécifications $SPEC_0$ qui n'utilisent aucune autre spécification. Un choix possible aurait été de choisir $\mathcal{M}_0$ égal à la catégorie des $SPEC_0$-algèbres satisfaisant la contrainte sur les générateurs. Toutefois ce choix n'est pas judicieux car il accepte en particulier des algèbres triviales. Pour pallier ce problème, la définition originale de la sémantique par classes stratifiées de modèles ne considérait que l'algèbre initiale dans ce cas. Cette définition présentait le défaut de n'accepter que des spécifications possédant un modèle initial dans le cas de base, ce qui est clairement un peu restrictif. C'est pourquoi nous avons étendu cette définition, en considérant des modèles dit « minimaux »(cf. chapitre 5).

L'approche ainsi définie peut sembler totalement indépendante de la sémantique et des spécifications choisies. On pourrait penser a priori qu'elle peut s'appliquer aussi bien à des spécifications observationnelles ou à des exception-spécifications qu'à la sémantique classique. Ce n'est malheureusement pas le cas.

Par exemple, lorsque nous avons développé une théorie de l'observabilité (chapitre 5), il s'est avéré que la contrainte sur les générateurs n'était pas directement applicable. Considérons à nouveau l'exemple de la réalisation des piles par tableaux et entiers (page 36), les piles étant abstraitement spécifiées comme en figure 6.1. Supposons par exemple que, lors de la création d'un tableau, celui-ci soit uniformément rempli de la valeur 0. Dès lors, tous les couples (t, h) obtenus par application des générateurs *empty* et *push* seulement possèdent la propriété suivante :

$$(\forall i \in I\!N)(\ i \geq h \Longrightarrow t[i] = 0\ )$$

ce qui n'est pas le cas d'une pile de la forme $pop(push(1, P))$ par exemple, puisque $t[h] = 1$ dans ce cas. Il en résulte que cette algèbre, bien qu'étant clairement un modèle observationnel acceptable des piles, n'est pas finiment engendrée par les générateurs *empty* et *push*. Pour obtenir une sémantique modulaire des spécifications observationnelles, il a donc fallu ne donner qu'une contrainte plus faible : toute valeur doit être *observationnellement* égale à un terme construit sur les générateurs. On voit donc là encore que l'étude d'exemples de théories de types abstraits algébriques peut conduire à modifier des définitions qui, au niveau « métathéorique » de la sémantique par classes stratifiées de modèles, semblaient parfaitement générales.

On peut aussi remettre en cause la contrainte de protection des modèles utilisés (i.e. des modèles de $\mathcal{M}_1$). Pour ce faire, considérons des spécifications avec traitement d'exceptions. Par exemple celle des piles avec traitement d'exceptions donnée en figure 6.2. (L'argumentation

---

7— Piles avec traitement d'exceptions

Spec « STACK ».
S : $\{Stack\}$
Σ : $empty :\rightarrow Stack$
$push : Nat \times Stack \rightarrow Stack$
$pop : Stack \rightarrow Stack$
$top : Stack \rightarrow Nat$
L : $\{Ok,\ UnderflowError,\ AccessError\}$
Ok-generators :
$empty \in Ok$
$P \in Ok \wedge e \in Ok \Longrightarrow push(e, P) \in Ok$
Ok-axioms :
$pop(push(e, P)) = P$
$top(push(e, P)) = e$
Exception handling :
$pop(empty) \in UnderflowError$
$top(empty) \in AccessError$

---

que nous développerons sera suffisamment simple pour qu'il ne soit pas nécessaire ici de définir totalement la sémantique des exception-algèbres). On remarque que $top(empty)$ donne lieu à une nouvelle valeur de sorte Nat, qui est une valeur erronée, mais qui n'en reste pas moins un junk. En effet, lors de la spécification des entiers naturels, l'erreur $top(empty)$ n'a pas été prise en compte. Il n'est pas pensable de prévoir, dès la spécification de Nat, toutes les erreurs possibles induites par des modules de spécification écrit ultérieurement. Des erreurs telles que $pred(0)$ ont été spécifiées, mais il n'y a aucune raison pour rendre $top(empty)$ égal à l'une de ces erreurs : on peut vouloir la récupérer ultérieurement par exemple, sans pour autant récupérer les entiers négatifs. En conséquence, il faut là encore adapter de manière ad hoc l'une des contraintes de la sémantique par classes stratifiées de modèles, à savoir la contrainte de protection des algèbres utilisées. Dans le cas présent, il suffit d'accepter les junks sous réserve qu'ils soient erronés (i.e. répondent à un label représentant un message d'erreur, et ne soient pas récupérés).

Une solution probablement judicieuse serait une définition « paramétrée »de la sémantique par classes stratifiées de modèles via des contraintes adaptables à la théorie des types abstraits algébriques particulière mise en jeu. Nous ne sommes cependant pas à l'abri d'autres modifications éventuelles. Par exemple cette sémantique a été utilisée avec succès pour bâtir une théorie de la réutilisation de (modules de) logiciels, mais il a fallu pour ce faire remettre en cause l'usage de *foncteurs* pour décrire la sémantique d'un module de spécification, et l'étendre via l'usage de

simples *applications*. Savoir laquelle de ces deux approches est la meilleure est encore un problème ouvert.

6.3 Le paradigme « Institution Independent »

Cette sous-section mentionne brièvement quelques autres obstacles rencontrés lorsque nous essayons de tirer profit de métathéories telles que celle des institutions. Elle ne doit surtout pas être considérée comme un réquisitoire contre ces métathéories, mais au contraire comme un embryon d'investigation vers des extensions ou aménagements éventuellement souhaitables.

Nous avons remarqué à deux reprises que la « condition de satisfaction »requise par les institutions n'est pas toujours facile à satisfaire pour des sémantiques dédiées. De plus, du fait que la sémantique par classes stratifiées de modèles ne requiert pas cette propriété[43], nous sommes assez peu motivés pour modifier ces sémantiques afin de les couler dans le moule des institutions. Pour être honnête, il faut remarquer que la théorie des institutions est en fait très souple. Les « contre-exemples » à la condition de satisfaction que nous avons relevés supposent que l'inclusion de signatures est directement un morphisme de signatures. Il est probablement possible d'enrichir la catégorie des signature pour y inclure des restrictions qui excluent les « mauvais » morphismes de signatures. Il n'en reste pas moins que, du point de vue d'un « utilisateur naïf » de cette métathéorie, il semble assez peu naturel d'inclure de telles restrictions.

Parmi les métathéories concurrentes, celle des « specification logics »proposée très récemment[44] par Ehrig, Baldamus et Orejas [EBO91] ne requiert qu'une seule des deux implications de la condition de satisfaction :

$$A \models \varphi \Longrightarrow U(A) \models \varphi$$

Pour être exact, les specification logics ne considère en fait aucunement la relation de satisfaction « $\models$ ». Elles ne traitent que des spécifications complètes et leurs catégories de modèles associées. La condition précédente est donc exprimée par :

$$A \in Alg(SPEC_2) \Longrightarrow U(A) \in Alg(SPEC_1)$$

pour toutes spécifications telles que $SPEC_1 \subseteq SPEC_2$. En conséquence, la sémantique observationnelle décrite en section 4 n'est pas non plus une specification logic. Par contre, on constate dans le chapitre 6 que les algèbres étiquetées forment une specification logic. Nous n'entrerons pas dans les détails ici, mais mentionnons toutefois que deux obstacles sont révélés par les algèbres étiquetées. L'usage des specification logics ne devient réellement fructueux que lorsque la théorie considérée possède une propriété dite « d'amalgamation », ou à défaut une propriété dite « d'extension » ; or les algèbres étiquetées ne possèdent pas ces propriétés sans conditions restrictives additionnelles. De plus, toutes les applications proposées par les auteurs des specification logics pour la modularité des spécifications sont fondées sur des foncteurs préservant les modèles utilisés (c'est-à-dire, comme pour la sémantique par classes stratifiées de modèles, les foncteurs F tels que $U(F(A)) = A$). Or nous avons remarqué que pour le traitement d'exceptions cette propriété ne traduit pas une bonne contrainte sémantique de protection des modèles à cause des junks produits par les valeurs erronées.

Mentionnons enfin que lorsque nous tentons d'étendre la théorie du test à des spécifications formelles quelconques (c'est-à-dire non nécessairement des spécifications algébriques simples au sens du groupe ADJ), nous avons clairement besoin d'une représentation formelle de ce qu'est une

[43] Elle requiert seulement l'existence d'un foncteur d'oubli relativement aux *signatures*

[44] si récemment que nous n'avons aucune traduction française viable de « specification logic » à proposer.

variable dans un axiome, et aussi d'une représentation formelle précise de l'usage des substitutions de variables dans la définition de la relation de satisfaction (cf. section 3). Ceci nous permettrait de définir ce que nous appelons un *jeu de tests exhaustif* de référence, au sein duquel nous pouvons sélectionner un jeu de tests adéquat. Cette tentative est relatée dans l'article « *A formal approach to software testing* »[45] (non reproduit dans ce document). Il s'avère alors que, d'une part, nous n'avons pas besoin de la condition de satisfaction (même sous une forme affaiblie), et d'autre part la relation de satisfaction n'est pas définie avec suffisamment de détail dans la théorie des institutions pour que nous puissions construire un jeu de tests exhaustif. Il semble que la « partie manquante » entre les institutions et notre théorie du test soit une modélisation systématique des notions de variables et de substitutions. Bien évidemment, a fortiori, les specification logics ne nous sont d'aucune utilité dans ce cadre puisqu'elles ne mentionnent même pas la relation de satisfaction.

D'un autre coté, nous avons remarqué qu'il était fortement souhaitable d'attacher une notion d'observabilité à une spécification entière, et pas seulement à un axiome. Nous avons alors soulevé le fait que « $A \models \varphi$ » *dépend de la spécification* à laquelle φ se réfère, puisque la relation de satisfaction est définie en fonction de l'observabilité définie globalement pour la spécification. De ce point de vue, l'approche des specification logics est préférable puisqu'elle traite une spécification dans son entier (via $A \in Alg(SPEC)$).

Plus généralement, il semble en fait que, pour des applications dans le domaine des types abstraits algébriques, les métathéories existantes ne modélisent pas avec suffisamment de précision les rapports entre « algèbres » et « axiomes » dans le contexte d'une spécification particulière. Il est clair qu'une meilleure modélisation abstraite de ce qu'est une variable dans un axiome nous serait d'une utilité majeure ; et qu'une modélisation abstraite de ce qu'est une substitution acceptable dans un cadre donné (i.e. à observation fixée, ou avec un certain traitement d'exceptions déclaré par ailleurs, etc.) permettrait d'envisager des applications plus fines. De plus, dans le cadre des types abstraits algébriques, une algèbre est toujours « un ensemble muni de... » et une métathéorie qui se contente de mentionner qu'une algèbre est un objet d'une catégorie quelconque laisse du même coup une trop grande latitude, qui nuit à l'expression de phénomènes propres aux spécifications algébriques.[46]

7 Conclusion

Slogan : *Testing, testing, testing...*

L'idée défendue dans ce document est que le développement d'un logiciel, de l'expression des besoins à la réalisation finale, peut être nettement amélioré par une étape intermédiaire de *spécifications formelles*. Nous avons choisi d'étudier l'une des approches possibles des spécifications formelles, à savoir les *spécifications algébriques*. Ces dernières sont bien adaptées pour refléter les aspects fonctionnels et modulaires du génie logiciel. Notre but est de contribuer à ce que les types abstraits algébriques ne soient pas seulement un outil théorique de modélisation de structures de données abstraites, mais rendent aussi de nombreux services en pratique, aussi bien lors de la phase de spécification/validation que lors de la phase de réalisation/vérification.

Durant la phase de réalisation/vérification, des techniques *d'implémentation abstraite* permettent d'exprimer avec précision des choix de mise en œuvre et de démontrer formellement

[45] Paru dans les Proc. AMAST-2, Iowa, May 1991 ; à paraître dans LNCS.

[46] Il est probable que le lecteur « reste sur sa faim » à la lecture de cette dernière sous-section, mais nous n'avons malheureusement pas (pas encore ?) de solution de rechange à proposer. Les commentaires que nous avons formulés sont même encore trop incomplets, à mon sens, pour se lancer dès maintenant dans une telle aventure...

leur correction. Par implémentations abstraites successives, on peut ainsi arriver à spécifier un découpage en modules élémentaires simples, faciles à coder. Il n'en reste pas moins que le logiciel final n'est pas directement modélisable dans un cadre homogène de types abstraits algébriques. Il en résulte donc que la vérification ne serait pas complète si l'étape de codage n'était pas elle même vérifiée. Des techniques de preuve de programme existent ; nous ne les avons pas abordées ici, mais nous avons remarqué que même si les sources d'un programme ont été prouvées, ceci ne dispense pas de tester. La raison en est qu'un *système* logiciel n'est pas un objet entièrement modélisable mathématiquement. Nous avons alors développé une théorie de la sélection de jeux de tests à partir de spécifications algébriques. Cette théorie permet d'exprimer formellement une évaluation de la « qualité » des jeux de tests sélectionnés. Dans le cadre du test, il n'est pas possible de démontrer la correction, mais nous pensons qu'il est fondamental de *savoir* ce qui ne l'a pas été. La « qualité » d'un jeu de test est donc exprimée sous forme d'*hypothèses* qui définissent « ce que le test ne démontre pas ». Mieux : on peut sélectionner automatiquement des jeux de tests à partir d'hypothèses standard bien choisies.
Il en résulte que nous avons non seulement établi que la théorie des types abstraits algébriques permet de modéliser formellement la phase de réalisation/vérification, mais aussi qu'elle peut fournir des techniques réellement exploitables pour cette phase.

Durant la phase de spécification/validation, il est important d'atteindre un style de spécification clair et concis, et de ne décrire que le « quoi »et non le « comment ». Nous avons remarqué sur quelques exemples que le pouvoir d'expression offert par la théorie (maintenant classique) proposée par le groupe ADJ était insuffisant. Nous avons principalement concentré nos efforts sur deux types de sémantiques qui résolvent des aspects importants des spécifications algébriques (l'*observabilité* et le *traitement d'exceptions*). La sémantique de l'observabilité que nous avons proposée permet à la fois de mieux modéliser la classe des réalisations correctes d'une spécification et d'atteindre un style de spécification réellement abstrait et concis. Motivés par le traitement d'exception, nous avons défini la théorie des *algèbres étiquetées* qui permet d'affiner la notion de satisfaction d'un axiome en limitant avec pertinence les substitutions de variables. Cette théorie est en fait très générale, et est un outil pour aborder de nombreuses applications. Elle a déjà été appliquée avec succès aux spécifications algébriques avec traitement d'exceptions (les exception-algèbres). Il était particulièrement important de proposer une bonne sémantique du traitement d'exception. Sans cela, les cas exceptionnels étaient arbitrairement ignorés (par exemple les structures bornées) ou récupérés sur des valeurs purement arbitraires (par exemple $top(empty) = 0$), conduisant à des spécifications peu crédibles.
Grâce à de telles extensions, nous franchissons un pas de plus vers un style de spécification clair et concis. Une meilleure lisibilité évite de nombreuses erreurs de spécification et nous pensons que la sémantique uniforme des algèbres étiquetées devrait faciliter la validation.

Nous avons aussi remarqué qu'un style de spécification réellement abstrait ne peut être atteint que si la sémantique des spécifications *modulaires* repose sur des *contraintes sémantiques implicites*. Ceci nous a donc motivés pour étudier des sémantiques modulaires adaptées aux spécifications observationnelles et aux spécifications étiquetées. Nous avons alors constaté que des « métathéories », qui d'un point de vue formel semblaient universelles, ne pouvaient pas être appliquées directement. Il faut, cas par cas, modifier certaines définitions ou renoncer à certaines propriétés qui semblaient pourtant parfaitement générales.

En ce sens, nous avons joué ici aussi un rôle de *testeurs*... de métathéories ; d'où le « slogan » énoncé au début de cette section. Plus précisément, pour chaque objet ou phénomène informatique, nous devons souvent faire face à un choix énorme de définitions ou énoncés a priori possibles. Il nous semble clair qu'une bonne démarche est de « tester »tout énoncé général par des exemples. C'est pour cette raison que, tout au long des articles inclus dans ce document,

nous avons toujours été très attentifs à étudier de nombreux exemples. Ces exemples sont pour nous indispensables. Ils constituent en fait le seul moyen de valider ou invalider une théorie.
La généralité des outils que nous manipulons, principalement issus de la théorie des catégories, masque souvent des phénomènes indésirables du point de vue du génie logiciel. Il faut donc toujours considérer avec une grande prudence les résultats généraux que nous pouvons énoncer et démontrer. En dépit de l'intuition « bien naturelle » qu'ils semblent refléter, ils ne sont pas nécessairement une modélisation fidèle des phénomènes que nous voulons traiter. Malgré le niveau d'abstraction auquel nous nous plaçons, il ne faut pas oublier que l'informatique est *aussi* une science expérimentale.

Les travaux commentés ici ne sont clairement qu'une « photographie instantanée ». Il est certain que de nombreuses extensions et améliorations restent à étudier. Par exemple :

– Nous avons mentionné que l'implémentation abstraite devrait être grandement facilitée en l'étudiant dans le cadre de sémantiques observationnelles. Les recherches exposées ici sur l'observabilité ne sont que des premières propositions. Le but à court terme est d'exploiter leurs implications sur l'implémentation abstraite ainsi que sur l'étape d'oracle du test.
– Il reste à étudier en détail comment « spécifier une sémantique observationnelle » au sein des algèbres étiquetées. Nous avons déjà effectué quelques investigations en ce sens, et nous avons remarqué qu'il était aisé de spécifier les termes observables.
– Bien d'autres extensions et résultats pour les algèbres étiquetées restent à étudier, tels le passage à l'ordre supérieur, un ensemble de règles d'inférences correct et complet, etc.
– Dans le domaine du test de logiciel, comme nous l'avons déjà mentionné, il reste à définir une approche étendue à des institutions quelconques (ou autres métathéories). Nous avons en effet constaté qu'avec l'approche actuelle, la propriété de non biais ne peut pas être traitée de manière indépendante de la théorie considérée. Cette généralisation posera assurément des problèmes non triviaux (cf. section 6.3). Pour ce faire, fidèles à notre approche par l'exemple, il sera sans doute fructueux d'étudier d'abord diverses applications du test à d'autres théories de spécification, comme par exemple la logique temporelle (pour le moment, nous ne testons aucune propriété faisant intervenir le temps).

Plus généralement, les travaux exposés ici montrent que les spécifications formelles peuvent apporter de nombreuses avancées dans le domaine du développement de logiciels. Elles sont non seulement le seul moyen d'exprimer formellement ce qu'est la correction d'un logiciel, mais aussi, en dehors de toute considération purement théorique, elle permettent de développer des techniques facilitant la production de logiciels de qualité. Tout sujet de recherche qui peut appuyer cette affirmation est a priori intéressant.

Références

[BCGKS87] Bidoit M., Capy F., Choppy C., Choquet N., Gresse C., Kaplan S., Schlienger F., Voisin F. : « *ASSPRO : un environnement de programmation interactif et intégré.* »Techniques et Sciences Informatiques (TSI), AFCET-Bordas, Vol.6, No.1, p.21-40, 1987.

[CH85] Coquand T., Huet G. : « *Constructions : a higher order proof system for mechanizing mathematics.* »EUROCAL 85, Linz, Springer-Verlag LNCS 203, 1985.

[Bid89] Bidoit M. : « *Pluss, un langage pour le développement de spécifications algébriques modulaires.* »Thèse d'Etat, University of Paris-Sud, 1989.

[EBO91] Ehrig H., Baldamus M., Orejas F. : « *New concepts for amalgamation and extension in the framework of specification logics.* »Research report Bericht-No 91/05, Technische Universitat Berlin, May 1991.

[EKMP82] Ehrig H., Kreowski H., Mahr B., Padawitz P. : « *Algebraic implementation of abstract data types.* »Theoretical Computer Science 20, 1982, pp.209-263.

[Fri85] Fribourg L. : « *SLOG : a logic programming language interpreter based on clausal superposition and rewriting.* »Proc. IEEE Symposium on Logic programming, Boston, July 1985.

[Fri90] Fribourg L. : « *Extracting Logic Programs from Proofs that use Extended Prolog Execution and Induction.* »7th Intl. Conf. on Logic Programming, Jérusalem, Juin 1990.

[GB84] Goguen J.A., Burstall R.M. : « *Introducing institutions.* »Proc. of the Workshop on Logics of Programming, Springer-Verlag L.N.C.S. 164, pp.221-256, 1984.

[GH86] Geser A., Hussmann H. : « *Experiences with the RAP system – a specification interpreter combining term rewriting and resolution techniques.* »Proc. ESOP 86, LNCS 213, p.339-350, 1986.

[GHW85] Guttag J.V., Horning J.J., Wing J.M. : « *LARCH in five easy pieces.* »Technical Report 5, Digital Systems Research Center, 1985.

[Gre84] Gresse C. : « *Contribution à la programmation automatique. CATY : un système de construction assistée de programmes.* »Thèse d'Etat, Université de Paris XI, Orsay, France, Mars 1984.

[GTW78] Goguen J.A., Thatcher J.W., Wagner E.G. : « *An Initial Algebra Approach to the Specification, Correctness, and Implementation of Abstract Data Types.* »Current Trends in Programming Methodology, ed. R.T. Yeh, Printice-Hall, Vol.IV, pp.80-149, 1978. (Also IBM Report RC 6487, October 1976.).

[Hen90] Hennicker R. : « *Context induction : A proof principle for behavioural abstractions and algebraic implementations.* »Technical Report MIP-9001, Fakultat fur Mathematik und Informatik, Universitat Passau, 1990.

[McL71] Mac Lane S. : « *Categories for the working mathematician.* »Graduate texts in mathematics, 5, Springer-Verlag, 1971.

[MN90] McMorran M.A., Nicholls J.E. : « *Z user manual.* »Technical Report TR12.274, IBM United Kingdom Laboratories Ltd, Hampshire, 1990.

[MSS89] Manca V., Salibra A., Scollo G. : « *Equational Typed Logic.* »Theoretical Computer Science, Vol.77, p.131-149, 1990. Also : Technical Report of the university of Twente, the Netherlands, Memoranda Informatica 89-43, July 1989.

(Une bibliographie plus détaillé sur les sujets abordés est fournie en fin de chaque chapitre.)

Chapitre 2 :
Correctness proofs for abstract implementations

Gilles BERNOT

LIENS, CNRS URA 1327
Ecole Normale Supe'rieure,
45 Rue d'Ulm,
F-75230 PARIS Ce'dex 05,
FRANCE

bitnet: bernot@frulm63
uucp: bernot@ens.ens.fr

(Appeared in Information and Computation, Vol.80, Num.2, pp.121-151, Feb. 1989.)

Contents

1 Introduction **47**

2 Problems raised by abstract implementation **49**

2.1 The abstraction function 49

2.2 The representation function 50

2.3 Reusability issues 51

3 Overview of our formalism **52**

4 The textual level **54**

4.1 The representation 54

4.2 The synthesis operations 55

4.3 The hidden component 56
4.4 The operation-implementing axioms 56
4.5 The equality representation 56

5 The presentation level **57**

6 The semantical level **60**

7 Correctness proofs **61**
7.1 Operation completeness 62
7.2 Data protection 63
7.3 Validity 64
7.4 Consistency 66
7.5 Correct implementations 68

8 Reuse of abstract implementations **69**
8.1 Implementations and enrichments 69
8.2 Composition of abstract implementations 71

9 Conclusion **71**

Abstract

New syntax and semantics for implementation of abstract data types are presented in this paper. This formalism leads to a simple, exhaustive description of the abstract implementation correctness criteria. These correctness criteria are expressed in terms of *sufficient completeness* and *hierarchical consistency*. Thus, correctness proofs of abstract implementations can be handled using classical tools such as *term rewriting* methods, *structural induction* methods or *syntactical methods* (e.g. fair presentations). The main idea of this approach is a fundamental distinction between *descriptive* and *constructive* specifications, using both abstraction and representation functions. Moreover, we show that the *composition* of several correct abstract implementations is always correct. This provides a formal foundation for a methodology of program development by stepwise refinement.

Key-words: abstract data types, abstraction, correctness proofs, implementation, initial model, mathematical programming, representation, theorem proving.

1 Introduction

For about twelve years [LZ75], [Gut75], [ADJ76], the formalism of abstract data types has been considered a major tool for writing hierarchical and modular specifications. Algebraic specifications provide the user with legible and relevant properties concerning the specified data structures. Nevertheless, as algebraic specifications give a *description* of the data structure *properties*, they should not provide the designer with a *constructive* specification of the corresponding *implementation*. To implement a data structure, the descriptive specification is not directly used. Rather, "resident" data structures (which have been previously implemented) are used. For instance, we implement a $STACK$ data structure by means of $ARRAY$. The following example shows the difference between "*descriptive*" and "*constructive*" specifications:

Example 1 : Let us specify stacks of natural numbers, $STACK(NAT)$, as follows:

$$pop(empty) = empty$$
$$pop(push(n, X)) = X$$
$$top(empty) = 0$$
$$top(push(n, X)) = n$$

This specification is *descriptive*, as it describes the basic properties of stacks. But this data structure is more efficiently implemented by means of arrays. A stack is then characterized by an array, which contains the elements of the stack, and an integer, which is the height of the stack. Without leaving off the abstract data type formalism, a *constructive* specification of the *implementation* of $STACK(NAT)$ using $ARRAY$ and NAT can be done as follows:

$$empty =< t, 0 >$$
$$push(n, < t, i >) =< t[i] := n \ , \ succ(i) >$$
$$pop(< t, 0 >) =< t, 0 >$$
$$pop(< t, succ(i) >) =< t, i >$$

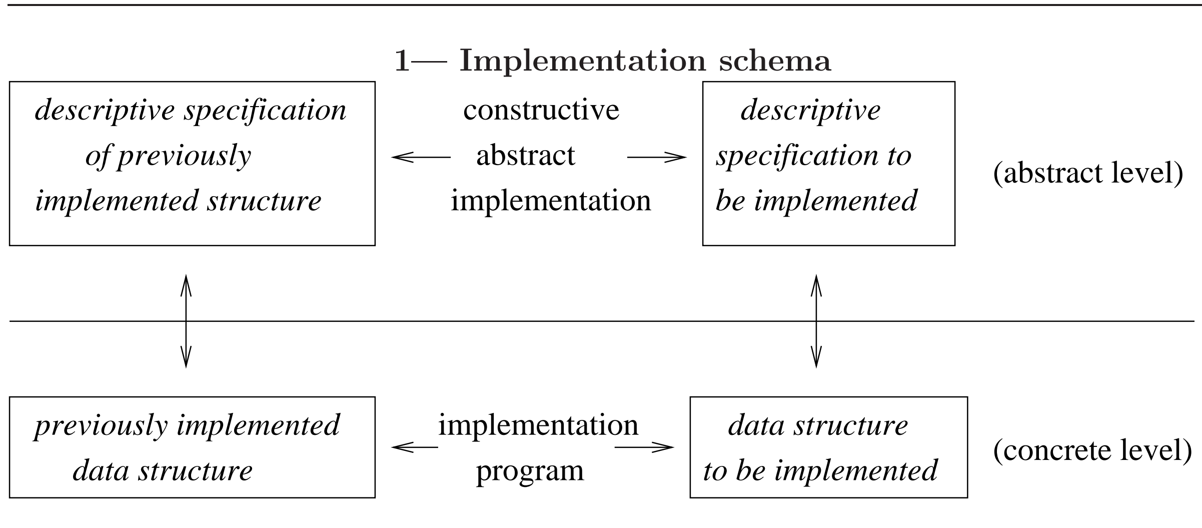
$top(< t, 0 >) = 0$
$top(< t, succ(i) >) = t[i]$

The first element pushed onto the stack is then $t[0]$; and the index i points to the place where the next element will be pushed (see [BBC86] for a more realistic treatment of the exceptional cases $pop(empty)$ and $top(empty)$).

However, we have to prove that the second set of (constructive) axioms is *correct* with respect to the data structure described by the first one.

It is well known that this need of establishing the correctness of an implementation with respect to the "designer's intentions" induces very difficult problems. The use of formal specifications (in particular algebraic specifications) is particularly fruitful, both for proving ([Hoa72], [ADJ78], [EKP80], [EKMP82], [SW82], [San87], [Sch87], [GM88]...) or for testing ([Bou82], [BCFG86], [Gau86]...) the correctness of an implementation. A natural idea is to describe the implementation problem in a homogeneous abstract specification framework; this leads to the concept of *abstract implementation*. We then hope that the usual proof techniques of abstract data types would facilitate correctness proofs of abstract implementations.

The situation can be outlined as in Figure 1.



Correctness proofs of abstract implementations can be done by using the notions of *representation invariants* and *equality representation* [GHM76], [Gau80]. For instance, the equality representation of Example 1 can be stated by:

$$< t, i > = < t', i' > \quad \text{if and only if} \quad i = i' \ \text{ and } \ t[j] = t'[j] \ \text{ for all } j = 0..i$$

Unfortunately, equality representation must be specified by the user, and nothing proves that it is correct. In particular, if we specify an equality representation where "everything is true," then every implementation will be correct. Since 1980, several works have formalized the notion of implementation correctness [EKP80], [EKMP82], [SW82], [San87], [Sch87] without using an explicit equality representation. All these works give *pure semantical* correctness criteria (such as existence of a morphism between two algebras).

Unfortunately, pure semantical correctness criteria do not provide the specifier with *theorem proving* methods (e.g. structural induction). It is therefore necessary to complete the abstract data type framework with an abstract implementation formalism which provides the user with "simple" correctness proof criteria.

In this paper, a new formalism for abstract implementations is provided. This formalism leads in a natural way to an exhaustive description of the abstract implementation correctness criteria. These correctness criteria can be checked via classical methods since they are expressed by means of *sufficient completeness* and *hierarchical consistency*. These two concepts are well known in classical abstract data types; thus we show that the *correctness* of abstract implementations does not require new concepts in the abstract data type field. This approach is especially powerful, since it is then possible to prove the correctness of an implementation using term rewriting techniques, structural induction, etc. Moreover, this formalism is compatible with *enrichment*, and the *composition* of two correct implementations always gives a correct result. This new definition of abstract implementation allows for the use of *positive conditional axioms*. We will show that this feature requires an equality representation, which in turn facilitates the correctness proofs. Moreover, the adequacy of the specified equality representation will be implied by the correctness of the implementation; thus the difficulties raised by the equality representation, which have been pointed out above, will be solved in this framework. Finally, the semantical level is very simple because it only uses the classical forgetful and synthesis functors. Thus, this formalism can easily be extended, for instance to algebraic data types with exception handling features [Ber86].

The next section presents the classical problems related to abstract implementation. Section 3 describes the main ideas of our formalism which solve these problems. Sections 4 through 6 describe our abstract implementation formalism. In Section 7, we show how correctness proofs of abstract implementation can be handled. Finally, we prove that abstract implementations cope with *enrichment* and *composition* (Section 8). We assume that the reader is familiar with elementary results of category theory and abstract data type theory.

2 Problems raised by abstract implementation

Abstract implementations are usually specified either with an *abstraction* function (presented in [Hoa72]), or with a *representation* function ([ADJ78], section 5.4.2).

2.1 The abstraction function

The abstraction takes previously implemented objects (e.g. arrays and natural numbers), and returns objects to be implemented (e.g. stacks). This is done by means of an *abstraction operation* (e.g. $A: \ ARRAY\ NAT \rightarrow STACK$). For instance, we obtain the axioms of the implementation of stacks by substituting $A(t,i)$ for $< t,i >$ in Example 1. Another trivial example is the following:

Example 2 : Natural numbers can be implemented by means of integers as follows:

$$0_{\mathbb{N}} = A(0_{\mathbb{Z}})$$
$$succ_{\mathbb{N}}(A(z)) = A(succ_{\mathbb{Z}}(z))$$
$$eq?_{\mathbb{N}}(A(z), A(z')) = eq?_{\mathbb{Z}}(z, z')$$

where $A: \ INT \rightarrow NAT$ is the abstraction operation.

The abstraction viewpoint is generalized and formalized in [EKP80], [EKMP82], and is also underlying in [SW82], [San87].

Unfortunately, abstraction operations synthesize too many objects in the sorts to be implemented. For instance, $A(create, 4)$ does not implement any stack, because if the height of a stack is equal to 4, then the four first ranges of the corresponding array must be initialized. In the same way, $A(-1)$ does not implement any natural number. As shown in [EKMP82], this fact prevents the specifier from carrying out simple correctness proofs by theorem proving methods. For instance, one of the proofs required is the implementation consistency: two objects which are distinct with respect to the descriptive specification must be implemented by two objects (synthesized by abstraction operations) which are distinct with respect to the constructive specification. The only formal concept of abstract data types which can handle such a condition is the *hierarchical consistency*. Thus, it is necessary to put together the constructive specification of our implementation (Example 2) and the descriptive specification to be implemented (NAT). We obtain a specification that contains both the (constructive) abstract implementation and the descriptive specification to be implemented, and we can check whether this specification is hierarchically consistent over NAT. NAT is specified as follows:

$$eq?_{\mathbb{N}}(0_{\mathbb{N}}, 0_{\mathbb{N}}) = True$$
$$eq?_{\mathbb{N}}(\ 0_{\mathbb{N}}\ ,\ succ_{\mathbb{N}}(m)\ ) \ = \ False$$
$$eq?_{\mathbb{N}}(\ succ_{\mathbb{N}}(n)\ ,\ 0_{\mathbb{N}})\ = \ False$$
$$eq?_{\mathbb{N}}(\ succ_{\mathbb{N}}(n)\ ,\ succ_{\mathbb{N}}(m)\ ) \ = \ eq?_{\mathbb{N}}(n, m)$$

But we obtain:

$$True \ = \ eq?_{\mathbb{N}}(\ 0_{\mathbb{N}}\ ,\ 0_{\mathbb{N}}) \ = \ eq?_{\mathbb{N}}(\ 0_{\mathbb{N}}\ ,\ succ_{\mathbb{N}}(A(-1))\ ) \ = \ False\ .$$

Consequently, although it is clearly correct, we cannot prove the consistency of our implementation this way.

2.2 The representation function

The aim of a representation is to provide a composition of previously implemented operations (e.g. those of NAT and $ARRAY$) for every operation to be implemented (e.g. *empty*, *push*, *pop*, *top*). For instance, the representation associated with Example 1 is specified as follows:

$$\rho(empty) = < t, 0 >$$
$$\rho(push(n, < t, i >)) = < t[i] := n\ ,\ succ(i) >$$

$$\rho(pop(< t, 0 >)) =< t, 0 >$$
$$\rho(pop(< t, succ(i) >)) =< t, i >$$
$$\rho(top(< t, 0 >)) = 0$$
$$\rho(top(< t, succ(i) >)) = t[i]$$

where ρ is the *representation* function.

Since representation only gives a representation for each operation to be implemented, it does not create undesirable values in the sorts to be implemented. Unfortunately, it is very difficult to give an algebraic meaning to such axioms. This is due to the fact that “$< _, _ >$” has no real algebraic definition. Considering $< _, _ >$ as an operation, its signature is necessarily: $<, >$: $ARRAY\ NAT \rightarrow STACK$ because it takes an array and a natural number, and returns a stack (as we apply *pop* to $< t, i >$). Consequently, the signature of $< _, _ >$ is the same as the signature of the abstraction operation. Thus, the function ρ is useless (in fact ρ is the identity), because the operation $< _, _ >$ can simply be used as an abstraction operation, which simplifies the previous specification.

A second way of looking at the representation may be to consider two representation operations:

$$\rho_1 : STACK \rightarrow ARRAY$$

$$\rho_2 : STACK \rightarrow NAT$$

But $\rho_1(empty)$ must be specified as a particular array. If we specify that $\rho_1(empty)$ can be equal to any array ($\rho_1(empty) = t$ together with $\rho_2(empty) = 0$, as in the abstraction case), then all arrays will be collapsed, which results in inconsistencies. Thus, breaking the representation operation into several operations is not powerful enough. In fact, we will develop an abstract implementation formalism which uses both ρ and $< _, _ >$. The problems mentioned above are avoided by means of intermediate “constructive sorts.”

2.3 Reusability issues

Let us assume that the *stack* data structure is already implemented by means of *arrays* and *natural numbers*. A user of this data structure will probably include it in some other programs. At the specification level, this means that some presentations over $STACK$ will be specified (presentations over $STACK$ can be viewed as abstract programs). But the user should never have to know how the implementation is done. In other words, (s)he knows the *descriptive* specification of $STACK$, but (s)he does not know the *constructive* specification of its implementation. Thus, every proof concerning an enrichment is done with respect to the descriptive specification of $STACK$, but not with respect to the implementation specification. This does not prove that the composition of the $STACK$ *implementation* with the new enrichment gives the expected result.

A particular subproblem is the composition of several implementations (i.e. implementations which reuse other implementations). All correctness proofs of the second implementation are handled with respect to the descriptive specification of the first implemented data structure; they are not done with respect to the constructive specification of this first implementation. *A priori*, the composition of the first implementation and the second

one is not proved to be correct, even if these two implementations are separately proved correct. In our framework, enrichments and compositions of correct abstract implementations always give the expected (semantical) results. This feature was not provided in any of the previous works in this area.

In order to achieve this goal, a specification of the equality representation must be included into the implementation (at least as soon as we want to enrich this implementation by a presentation containing *conditional* axioms). For example, a presentation over $STACK$ can contain a conditional axiom of the form:

$$pop(X) = empty \Longrightarrow M = N$$

We may have: $X = push(n, empty)$. The implementations of the terms *empty* and $pop(push(n, empty))$ are then $< create, 0 >$ and $< create[0] := n\ ,\ 0 >$. These pairs are not equal, but the premise of this axiom must be satisfied. If the implementation cannot detect *when two distinct pairs implement the same stack*, then our enrichment viewed through the implementation will not be correct, since some instances of this axiom are not taken into account. Thus, it is necessary to include the equality representation into the implementation in order to handle conditional axioms of enrichments. We will show that equality representation is also a useful tool for correctness proofs.

3 Overview of our formalism

We have shown that the abstraction (A) has the advantage of *synthesizing* products of previously implemented sorts. Therefore, the inconsistencies described with ρ_1 and ρ_2 in section 2.2 are avoided. But abstraction leads to complicated correctness proofs because it adds some undesirable values in the sorts to be implemented. A *restriction* is necessary before defining implementation correctness. On the other hand, the representation (ρ) solves this problem, because it only returns the implementation of each values to be implemented. Intuitively, the image of ρ is just the result of the restriction. Representation automatically handles restriction. But we must face the difficulty of giving an algebraic syntax for representation. In fact, we will take advantage of both abstraction and representation by using intermediate *constructive sorts*. Indeed, the main idea of the abstract implementation formalism described here is a systematic distinction between *descriptive* and *constructive* specifications or models. A *descriptive* specification or model only results from the abstract description of the known or required properties of the data structure under consideration. A *constructive* specification or model results from information or choices about its implementation.

Let us state the problem as follows:

- The previously implemented data structure (e.g. NAT and $ARRAY$) is specified by $\mathbf{SPEC}_0 = (\mathbf{S}_0, \Sigma_0, \mathbf{A}_0)$, where $\mathbf{S}_0$ is a set of sorts, Σ_0 is a set of operations with arity in $\mathbf{S}_0$, and $\mathbf{A}_0$ is a set of *positive conditional axioms* over the signature $(\mathbf{S}_0, \Sigma_0)$. $\mathbf{SPEC}_0$ is called the *resident* specification.

  Of course, $\mathbf{SPEC}_0$ is a *descriptive specification.* $\mathbf{SPEC}_0$ does not explain how resident sorts are implemented; it only describes “what properties we know” about

the resident values. In particular the initial algebra $T_{\mathbf{SPEC}_0}$ is a "descriptive model" of resident values; $T_{\mathbf{SPEC}_0}$ does not necessarily reflect the constructive (previously completed) implementation of resident sorts.

- We want to implement a data structure described by $\mathbf{SPEC}_1 = (\mathbf{S}_1, \Sigma_1, \mathbf{A}_1)$.

  $\mathbf{SPEC}_1$ is only a *descriptive specification* of "what properties we want to obtain" after the implementation is performed (e.g. $NAT + STACK$). In particular, the $\mathbf{SPEC}_1$-initial algebra $T_{\mathbf{SPEC}_1}$ is only a "reference model" (for correctness) which does not necessarily reflect the actual implementation semantics. $T_{\mathbf{SPEC}_1}$ is a *descriptive model* of the expected implementation result.

- Notice that $\mathbf{SPEC}_0$ and $\mathbf{SPEC}_1$ are not necessarily disjoint. For example, NAT is a specification included both in $\mathbf{SPEC}_0 = NAT{+}ARRAY$ and in $\mathbf{SPEC}_1 = NAT + STACK$. In the following, we assume that $\mathbf{SPEC}_0$ and $\mathbf{SPEC}_1$ are both *persistent* (i.e. hierarchically consistent and sufficiently complete) over the common specification $\mathbf{SP}$:

$$\mathbf{SP} = (\mathbf{S},\Sigma,\mathbf{A}) = (\mathbf{S}_0 \cap \mathbf{S}_1 \ , \ \Sigma_0 \cap \Sigma_1 \ , \ \mathbf{A}_0 \cap \mathbf{A}_1).$$

The abstract implementation problem is to define a *constructive* specification of $\mathbf{SPEC}_1$ using $\mathbf{SPEC}_0$; and to provide the specifier with usable correctness criteria.

An *abstract implementation* will be performed in five steps, using intermediate *constructive sorts* containing *constructive values*:

- The first step describes the *representation.* For each (descriptive) sort of $\mathbf{SPEC}_1$ (e.g. $STACK$), there is a *constructive* sort which represents it ($\overline{STACK}$). Intuitively, $\overline{STACK}$ will be the product sort "Array × Natural." For each (descriptive) operation of $\mathbf{SPEC}_1$ (e.g. *empty*, *push*, *pop*, *top*), there is a *constructive* operation which is its *actual implementation* ($\overline{empty}$, $\overline{push}$, $\overline{pop}$, $\overline{top}$). These constructive operations work on the constructive sorts (e.g. $\overline{STACK}$) instead of directly working on the descriptive sorts to be implemented ($STACK$).

  Notice that there are also constructive operations and a constructive sort associated with NAT (as $NAT \subseteq \mathbf{SPEC}_1$). Since NAT has already been implemented (included in $\mathbf{SPEC}_0$), $\overline{NAT}$ is simply a *copy* of NAT. Intuitively, this corresponds to the following fact: $\mathbf{SPEC}_0$ is a *descriptive* specification; we do not know the constructive (previously completed) implementation of NAT. Consequently, *by default*, we synthesize $\overline{NAT}$ as a copy of NAT.

- The second step synthesizes the *constructive values* used by the implementation. These constructive values are generated by means of *synthesis operations.* For example, the synthesis operation associated with $\overline{STACK}$ is the abstraction operation $\langle _ , _ \rangle_{STACK} : \ ARRAY \ NAT \rightarrow \overline{STACK}$ that synthesizes the product sort $\overline{STACK}$ ($ARRAY \times NAT$), associated with $STACK \in \mathbf{S}_1$. Moreover, the synthesis operation associated with $\overline{NAT}$ is simply (by default) a copy operation $\langle _ \rangle_{NAT} : \ NAT \rightarrow \overline{NAT}$.

- The third step is only a convenient (hidden) enrichment of the previously synthesized data structure. This *hidden component* of the implementation was first introduced in [EKP80]. It allows us to add hidden operations which are useful to specify the

implementation. For instance, if the resident specification of integers (Example 2) does not contain the operation $eq?_{\mathbb{Z}}$, then it is very useful to define it in the hidden component before specifying the main part of the implementation.

- The fourth step is the usual constructive specification of the implementation. It recursively specifies the actual implementation of each new constructive operation ($\overline{empty}$,$\overline{push}$,...) on the constructive sorts ($\overline{STACK}$). This step is handled by means of conditional axioms, as in previous examples.
- The last step specifies the equality representation. It will be specified by means of a set of conditional axioms. Our last step specifies the implementation of the *classes* (or equivalently *values*) to be implemented; while the fourth step only specifies the implementation of *terms* to be implemented.

This new fundamental distinction between descriptive and constructive aspects will be reflected on three different levels: the *textual* level, the *presentation* level and the *semantical* level.

- the *textual level* (Section 4) only contains the informations that the specifier must provide in order to define the implementation
- the *presentation level* (Section 5) is automatically deduced from the textual level; it gives a complete algebraic specification for the implementation (which will be useful for correctness proofs)
- the *semantical level* (Section 6) is automatically deduced from the presentation level; it describes the models (algebras) of the implementation.

Similar levels have been first introduced by [EKP80]. They have been shown to be a firm basis to define correctness for abstract implementations.

4 The textual level

Definition 3 : (Textual level). We define an *abstract implementation of* $\mathbf{SPEC}_1$ *by* $\mathbf{SPEC}_0$, denoted by **IMPL**, as a tuple:

$$\mathbf{IMPL} = (\rho\ ,\ \mathbf{\Sigma}_{SYNTH}\ ,\ \mathbf{H}\ ,\ \mathbf{A}_{OP}\ ,\ \mathbf{A}_{EQ})$$

where ρ is the *representation*, $\mathbf{\Sigma}_{SYNTH}$ is the set of *synthesis operations*, $\mathbf{H}$ is the *hidden component*, $\mathbf{A}_{OP}$ is the set of *constructive axioms*, and $\mathbf{A}_{EQ}$ is the *equality representation.*

These five parts are precisely defined in the following subsections.

4.1 The representation

Definition 4 : The *representation*, ρ, is the signature isomorphism defined as follows:

- for each descriptive sort to be implemented, $s \in \mathbf{S}_1$, there is an associated *constructive sort,* $\overline{s}$. We denote the set of constructive sorts by $\overline{\mathbf{S}_1}$ (actual constructive values of sort $\overline{s}$ will be generated by the synthesis operations). Thus, $\overline{\mathbf{S}_1}$ is a copy of $\mathbf{S}_1$. The constructive sort $\overline{s}$ implements s.
- for each operation to be implemented, $op: \ s_1 \cdots s_n \rightarrow s_{n+1}$ $(\in \Sigma_1)$, there is a *constructive operation,* $\overline{op}: \ \overline{s_1} \cdots \overline{s_n} \rightarrow \overline{s_{n+1}}$, where $\overline{s_i}$ is the constructive sort associated with s_i. We denote the set of constructive operations by $\overline{\Sigma_1}$. The constructive operation $\overline{op}$ implements op.

ρ is the signature isomorphism from $(\mathbf{S}_1, \Sigma_1)$ to $(\overline{\mathbf{S}_1}, \overline{\Sigma_1})$. ρ is called the *representation signature isomorphism*, or simply the *representation*, since it gives the constructive representation of each descriptive sort/operation to be implemented. For instance, ρ sends the sort NAT to $\overline{NAT}$, $STACK$ to $\overline{STACK}$, $push: \ NAT \ STACK \rightarrow STACK$ to $\overline{push}: \ \overline{NAT} \ \overline{STACK} \rightarrow \overline{STACK}$, and so on.

Remark 5 : Notice that the representation ρ may seem useless. In practice, it is clear that we do not ask the specifier to explicitly characterize ρ . Nevertheless, on a theoretical viewpoint, it is necessary to precisely specify the correspondence between the descriptive signature to be implemented and its constructive implementation.

4.2 The synthesis operations

Definition 6 : The set of *synthesis operations*, denoted by $\mathbf{\Sigma}_{SYNTH}$ is defined as follows: for each constructive sort, $\overline{s} \in \overline{\mathbf{S}_1}$, there is a synthesis operation, $\langle \ldots \rangle_s: \ r_1 \cdots r_m \rightarrow \overline{s}$, where all the r_i are resident sorts in $\mathbf{S}_0$.

For instance, the synthesis operation associated with the sort $STACK$ is the "abstraction operation:" $\langle _, _ \rangle_{STACK}: \ ARRAY \ NAT \rightarrow \overline{STACK}$; the synthesis operation associated with NAT is the "copy operation:" $\langle _ \rangle_{NAT}: \ NAT \rightarrow \overline{NAT}$.

Remark 7 : The synthesis operation associated with each previously implemented sort of **SP** (e.g. NAT) will be a copy operation. Thus, in practice, we never ask the specifier to give the synthesis operation associated with these sorts. Nevertheless, this copy is useful, and necessary, when rigorously proving the correctness of an abstract implementation. Intuitively, the introduction of constructive sorts, together with the representation signature isomorphism, handles the *restriction* problem. This restriction must apply to all sorts to be implemented, including the sorts of **SP**. For example, $top(<create, 4>)$ could be a new value belonging to NAT, which must be removed before verifying the correctness of our $STACK$ implementation (as the pair $<create, 4>$ is not a reachable stack). With this abstract implementation formalism, $\overline{top}(<create, 4>)$ will be of *constructive* sort $\overline{NAT}$, and thus, the *descriptive* sort NAT is preserved.

4.3 The hidden component

Definition 8 : The *hidden component* of **IMPL**, $\mathbf{H} = (\mathbf{S}_H, \mathbf{\Sigma}_H, \mathbf{A}_H)$, is a presentation over **SORTimpl** = $\mathbf{SPEC}_0 + (\overline{\mathbf{S}_1}, \mathbf{\Sigma}_{SYNTH}, \emptyset)$ that enriches the synthesized data structures in order to facilitate the implementation.

In our $STACK$ by $ARRAY$ example, **H** is empty. An example of non empty hidden component is given in Example 11 (Section 4.5 below).

4.4 The operation-implementing axioms

Definition 9 : We denote by $\mathbf{A}_{OP}$ the set of constructive axioms of **IMPL**. $\mathbf{A}_{OP}$ is a set of positive conditional axioms over the signature

$$(\mathbf{S}_0 + \mathbf{S}_H + \overline{\mathbf{S}_1}\ ,\ \Sigma_0 + \mathbf{\Sigma}_{SYNTH} + \mathbf{\Sigma}_H + \overline{\Sigma_1})$$

It specifies the actual implementation of the constructive operations $\overline{op}$. $\mathbf{A}_{OP}$ is the set of *operation-implementing* axioms.

The axioms of $\mathbf{A}_{OP}$ are those specified for abstraction:

$$\begin{array}{l}
\overline{empty} = \langle t, 0\rangle_{STACK} \\
\overline{push}(\langle n\rangle_{NAT}, \langle t, i\rangle_{STACK}) = \langle t[i] := n\ ,\ succ(i)\rangle_{STACK} \\
\overline{pop}(\langle t, 0\rangle_{STACK}) = \langle t, 0\rangle_{STACK} \\
\overline{pop}(\langle t, succ(i)\rangle_{STACK}) = \langle t, i\rangle_{STACK} \\
\overline{top}(\langle t, 0\rangle_{STACK}) = \langle 0\rangle_{NAT} \\
\overline{top}(\langle t, succ(i)\rangle_{STACK}) = \langle t[i]\rangle_{NAT}
\end{array}$$

Of course, these axioms can always be automatically deduced from those of Example 1 (i.e. from axioms without "bars").

4.5 The equality representation

Definition 10 : The *equality representation*, denoted by $\mathbf{A}_{EQ}$, is a set of positive conditional axioms which can use all the sorts and operations previously mentioned: $(\mathbf{S}_0 + \mathbf{S}_H + \overline{\mathbf{S}_1} + \mathbf{S}_1\ ,\ \Sigma_0 + \mathbf{\Sigma}_H + \mathbf{\Sigma}_{SYNTH} + \overline{\Sigma_1} + \Sigma_1)$.

For instance, the equality representation of our $STACK$ by $ARRAY$ example can be specified as follows:

$$\langle t, 0\rangle_{STACK} = \langle t', 0\rangle_{STACK}$$

$$\left\{ \begin{array}{c} \langle t, i\rangle_{STACK} = \langle t', i\rangle_{STACK} \\ \wedge\ \ t[i] = t'[i] \end{array} \right\} \Longrightarrow \langle t, succ(i)\rangle_{STACK} = \langle t', succ(i)\rangle_{STACK}$$

(In fact, $\mathbf{A}_{EQ}$ can be empty in this example, since $\mathbf{A}_{OP}$ already implies our two axioms; but this is specific to the $STACK$ example).

Let us specify another standard example: the implementation of SET by $STRING$ (of natural numbers, for instance).

Example 11 : (textual implementation of SET by $STRING$). The representation signature isomorphism ρ sends the descriptive sorts $BOOL$, NAT and SET to the constructive sorts $\overline{BOOL}$, $\overline{NAT}$ and $\overline{SET}$ respectively; and sends $\emptyset$ to $\overline{\emptyset}$, ins to $\overline{ins}$, $\in$ to $\overline{\in}$, as well as $True$ to $\overline{True}$, $False$ to $\overline{False}$, and so on.

The synthesis operations of $\mathbf{\Sigma}_{SYNTH}$ are:

$\langle _ \rangle_{SET}$: $STRING \rightarrow \overline{SET}$ <!elem "true synthesis operation"
$\langle _ \rangle_{NAT}$: $NAT \rightarrow \overline{NAT}$ "copy operation"
$\langle _ \rangle_{BOOL}$: $BOOL \rightarrow \overline{BOOL}$ "copy operation"

If $STRING$ does not contain the operations $remove$ and $occurs$, then $\mathbf{H}$ may specify them as hidden operations:

$$
\begin{array}{l}
remove(x, \lambda) = \lambda \\
remove(x, add(x, s)) = s \\
eq?(x, y) = False \implies remove(x, add(y, s)) = add(y, remove(x, s)) \\
occurs(x, \lambda) = False \\
occurs(x, add(x, s)) = True \\
eq?(x, y) = False \implies occurs(x, add(y, s)) = occurs(x, s)
\end{array}
$$

From the second axiom, $remove(x, s)$ only removes the first occurrence of x in s. This does not matter because sets will only be represented by non-redundant strings.

The constructive axioms of $\mathbf{A}_{OP}$ are:

$$
\begin{array}{l}
\overline{\emptyset} = \langle \lambda \rangle_{SET} \\
occurs(x, s) = True \Longrightarrow \overline{ins}(\langle x \rangle_{NAT}, \langle s \rangle_{SET}) = \langle s \rangle_{SET} \\
occurs(x, s) = False \Longrightarrow \overline{ins}(\langle x \rangle_{NAT}, \langle s \rangle_{SET}) = \langle add(x, s) \rangle_{SET} \\
\overline{del}(\langle x \rangle_{NAT}, \langle s \rangle_{SET}) = \langle remove(x, s) \rangle_{SET} \\
\langle x \rangle_{NAT} \overline{\in} \langle s \rangle_{SET} = \langle occurs(x, s) \rangle_{BOOL}
\end{array}
$$

And the equality representation $\mathbf{A}_{EQ}$ is given as follows:

$$
\langle add(x, add(y, s)) \rangle_{SET} = \langle add(y, add(x, s)) \rangle_{SET}
$$

Notice that this axiom does not create inconsistency on strings, because it applies to $\overline{SET}$.

5 The presentation level

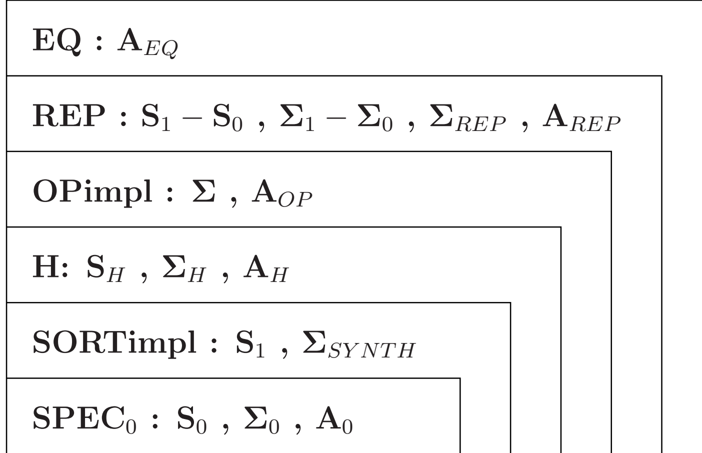
A *presentation* is automatically built from the textual level of an abstract implementation. This presentation is an enrichment of $\mathbf{SPEC}_0$. It is useful for proving the correctness of an implementation. Intuitively, all well known difficulties of abstract implementation are treated by the presentation level. These difficulties are mainly the *restriction* to reachable values, and the *identification* of several implementation values which represent the same object. In [EKP80], [EKMP82], [SW82], [San87], these two problems are handled at the

semantical level. This results in a rigorous definition of correctness, but does not provide the specifier with useful correctness proof tools (since correctness is mainly related to the existence of a morphism between two algebras). Here, the restriction problem is explicitly handled via the intermediate constructive sorts and the representation, while the identification problem is explicitly handled via the equality representation.

The *presentation level* associated with the textual level of an abstract implementation is defined as in Figure 5. where **SORTimpl** is a presentation over the specification

---

2— The presentation level



$\mathbf{SPEC}_0$, **H** is a presentation over the specification $\mathbf{SPEC}_0$+**SORTimpl** (union of $\mathbf{SPEC}_0$ and **SORTimpl**), and so on. These presentations can be explained as follows:

- $\mathbf{SPEC}_0$ is the *descriptive* specification of the *resident* (previously implemented) data structure.
- **SORTimpl** is the *synthesis* presentation. For each descriptive sort to be implemented $s \in \mathbf{S}_1$, the corresponding constructive sort $\overline{s} \in \overline{\mathbf{S}_1}$ is synthesized by means of the synthesis operations $\langle \ldots \rangle_s : r_1 \cdots r_m \rightarrow \overline{s}$. Moreover, **SORTimpl** does not contain any axiom. Thus, **SORTimpl** "*implements the constructive sorts*" as free products, or copies, of resident sorts. The initial algebra $T_{SORTimpl}$ contains the *available* constructive structure which our abstract implementation can use.
- **H** is the *hidden* presentation of the abstract implementation. **H** is a presentation over $\mathbf{SPEC}_0$+**SORTimpl**, as defined in previous section. It will facilitate the constructive specification of the abstract implementation by enriching the resident or available constructive specifications (cf. *remove* and *occurs* in Example 11).
- **OPimpl** is the *operation-implementing* part of the presentation level. It specifies how the constructive operations $\overline{op} \in \overline{\Sigma_1}$ (implementing the descriptive operations $op \in \Sigma_1$) work over the previously synthesized constructive sorts. This is done by means of the operation-implementing axioms $\mathbf{A}_{OP}$, as defined in the previous section. Thus, the initial algebra T_{OPimpl} handles the constructive implementation of the constructive operations ($\overline{op}$) over the synthesized sorts.

- **REP** is the *representation* presentation. It *explicitly* specifies (in the specification) the effect of the representation signature isomorphism (ρ). We will define $\mathbf{\Sigma}_{REP}$ and $\mathbf{A}_{REP}$ below.

  This presentation **REP** has two principal characteristics. First, it *syntactically* specifies the correspondence between the descriptive operations op and the constructive operations $\overline{op}$. Second, it explicitly handles the *restriction* part of the abstract implementation. Let us return to Example 2 (implementation of NAT using INT). With our new formalism, values such as $\langle -1 \rangle_{NAT}$ are not of sort NAT; they belong to $\overline{NAT}$ which is a copy of INT. There is no NAT-term t such that $\rho(t)$ is equal to $\langle -1 \rangle_{NAT}$.

- **EQ** is the *equality representation* part of the presentation level. It specifies when two distinct available constructive values represent the same descriptive value to be implemented. This is done via the set $\mathbf{A}_{EQ}$ of conditional axioms. In view of the definition of $\mathbf{A}_{EQ}$ given in previous section, **EQ** is a presentation over the signature $(\mathbf{S}_0 + \mathbf{S}_H + \overline{\mathbf{S}_1} + \mathbf{S}_1\ ,\ \Sigma_0 + \mathbf{\Sigma}_H + \mathbf{\Sigma}_{SYNTH} + \overline{\Sigma_1} + \Sigma_1)$; in particular **EQ** is a presentation over **SPEC**$_0$+**H**+**SORTimpl**+**OPimpl**+**REP**. Thus, the initial algebra T_{EQ} handles the *identification* of constructive values which represent the same descriptive value to be implemented.

$\overline{\mathbf{S}_1}$, $\mathbf{\Sigma}_{SYNTH}$, **H**, $\overline{\Sigma_1}$, $\mathbf{A}_{OP}$ and $\mathbf{A}_{EQ}$ are already defined in Section 4. $\mathbf{\Sigma}_{REP}$ and $\mathbf{A}_{REP}$ are defined as follows:

- $\mathbf{\Sigma}_{REP}$ is the set of *representation operations*. For each descriptive sort to be implemented, $s \in \mathbf{S}_1$, there is a representation operation: $\overline{\rho_s} :\ s \longrightarrow \overline{s}$.

- $\mathbf{A}_{REP}$ is the set of axioms which state that $\overline{\rho_s}$ extends the representation signature isomorphism ρ. This means that for each Σ_1-ground-term t of sort s, $\overline{\rho_s}(t)$ is equal to the $\overline{\Sigma_1}$-term deduced from t via ρ. Thus, for each operation to be implemented, $op \in \Sigma_1$, $\mathbf{A}_{REP}$ contains the following axiom:

$$\overline{\rho_s}(op(x_1, \cdots, x_n)) \ = \ \rho(op)(\overline{\rho_s i}(x_i), \cdots, \overline{\rho_{s_n}}(x_n))$$

  where s is the target sort of op, s_i is the sort of x_i , and $\rho(op)$ is equal to $\overline{op}$.

  Moreover, we have to specify that $\overline{\rho_s}$ and $\langle \ldots \rangle_s$ both work as *copy* operations on the signature of **SP** (common specification). Thus, for each sort s of **SP**, $\mathbf{A}_{REP}$ contains the following axiom:

$$< x >_s = \overline{\rho_s}(x)$$

  (It implies that $\overline{0} = \langle 0 \rangle_{NAT}$ and $\overline{succ}(\langle n \rangle_{NAT}) = \langle succ(n) \rangle_{NAT}$ in $\overline{NAT}$).

  Finally, $\mathbf{A}_{REP}$ contains the following axiom for each descriptive sort $s \in \mathbf{S}_1$:

$$\overline{\rho_s}(x) = \overline{\rho_s}(y) \Longrightarrow x = y\ .$$

  The intuitive meaning of this axiom is the following: if two (descriptive) terms to be implemented, x and y, are represented by the same constructive value ($\overline{\rho_s}(x) = \overline{\rho_s}(y)$), then they must be equal after the implementation is done ($x = y$). The reason why this constraint is required can be explained as follows: our goal is to describe the

data structure that "the user thinks (s)he manipulates" after the implementation is done. If the terms x and y get the same representation, then the user of the implementation cannot distinguish x from y ; consequently, "(s)he thinks that x is equal to y." Such an amalgamation is exactly handled by the axiom specified above.

Notice that $\mathbf{\Sigma}_{REP}$ and $\mathbf{A}_{REP}$ are automatically deduced from the signature isomorphism ρ . Thus the presentation level is always automatically built from the textual definition of **IMPL** without help from the specifier.

Example 12 : In the $STACK$ by $ARRAY$ example, $\mathbf{A}_{REP}$ is deduced from the signature isomorphism ρ as follows:

$$\overline{\rho_{STACK}}(empty) = \overline{empty}$$
$$\overline{\rho_{STACK}}(push(n, X)) = \overline{push}(\overline{\rho_{NAT}}(n), \overline{\rho_{STACK}}(X))$$
$$\overline{\rho_{STACK}}(pop(X)) = \overline{pop}(\overline{\rho_{STACK}}(X))$$
$$- - etc - -$$
$$\overline{\rho_{STACK}}(X) = \overline{\rho_{STACK}}(Y) \Longrightarrow X = Y$$
$$\overline{\rho_{NAT}}(m) = \overline{\rho_{NAT}}(n) \Longrightarrow m = n$$

Remark 13 : This specification, from $\mathbf{SPEC}_0$ to **OPimpl**, is very close to the "syntactical level" of [EKP80] or [EKMP82]. Our formalism mainly adds the presentations **REP** and **EQ**. It can be shown that **REP** explicitly specifies the *Restriction* functor of the [EKMP82] semantics; and when the abstract implementation is correct, **EQ** explicitly specifies the *Identification* functor of the [EKMP82] semantics.

6 The semantical level

We have shown the following: $T_{SORTimpl}$ contains all synthesized constructive sorts; T_{OPimpl} handles the constructive implementation of all constructive operations ($\overline{op}$); T_{REP} does not add unreachable values to the descriptive sorts to be implemented (thus *restriction* is already included in T_{REP}); and T_{EQ} contains the *identification* of constructive values which implement the same descriptive value. Consequently, T_{EQ} is not far from the semantical result of the abstract implementation.

Notice that T_{EQ} contains all intermediate sorts and operations used by the abstract implementation. But the user of the new implemented data structure must not use the specific operations and sorts of the implementation. It is necessary to forget: the resident sorts and operations which are not in $\mathbf{SPEC}_1$, the hidden sorts and operations, the intermediate constructive sorts, the synthesis operations, and the constructive operations $\overline{op}$. Then, we get a new Σ_1-algebra which contains only what the user "thinks (s)he manipulates." This "user view" algebra is called the semantical result of **IMPL** and is denoted by SEM_{IMPL}.

The semantics of an abstract implementation **IMPL** is the composition of two functors:

$$\begin{array}{ccccc} Alg(\mathbf{SPEC}_0) & \xrightarrow{\quad F_{\mathbf{SORTimpl+..+EQ}} \quad} & Alg(\mathbf{EQ} + .. + \mathbf{SPEC_0}) & \xrightarrow{\quad U_{\mathbf{\Sigma}_1} \quad} & Alg(\mathbf{S}_1, \mathbf{\Sigma}_1) \\ T_{\mathbf{SPEC}_0} & \longmapsto & T_{EQ} & \longmapsto & SEM_{IMPL} \end{array}$$

$F_{SORTimpl+..+EQ}$ is the usual synthesis functor associated with the presentation **SORTimpl+H+OPimpl+REP+EQ** over $\mathbf{SPEC}_0$ (left adjoint to the forgetful functor). U_{Σ_1} is the usual forgetful functor:

$$U_{\Sigma_1} \ : \ \text{Alg}(\mathbf{SPEC}_0\mathbf{+SORTimpl+H+OPimpl+REP+EQ}) \rightarrow \text{Alg}(\mathbf{S}_1, \Sigma_1, \emptyset)$$

So, SEM_{IMPL} describes the "*user view*" of the new implemented data structure. SEM_{IMPL} is the part of T_{EQ} corresponding to the descriptive sorts to be implemented ($\mathbf{S}_1$); and the only operations accessible to the user are those of Σ_1.

Notice that this semantical level is considerably simpler than the ones of [EKP80], [EKMP82], [SW82]... This reflects the fact that all abstract implementation problems (restriction and identification) are handled at the presentation level. Moreover, the next section shows that correctness criteria can be stated in a constructive manner, because restriction and identification are taken into account at the presentation level.

At first glance, our formalism may seem to be more restrictive than the [EKP80], [EKMP82] formalism, because we require a specification of the equality representation. The [EKP80], [EKMP82] formalism uses the equations of $\mathbf{A}_1$ in order to perform the Identification functor. In fact, when we choose $\mathbf{A}_{EQ} = \mathbf{A}_1$ in our formalism, we get exactly the same semantical result SEM_{IMPL}. The interresting point here is that our semantics avoids the Restriction functor by using the representation ρ (another title of this paper could be "Implementation without Restriction"...). All the difficulties encountered in [EKP80], [EKMP82] are due to this "bad" Restriction functor. Correctness proofs are considerably simpler, in the [EKMP82]]] formalism, when the Identification functor can be performed before the Restriction functor (*IR* semantics instead of *RI* semantics). It can be shown that the *IR* semantics of [EKMP82] is equivalent to the semantics obtained in our formalism by $\mathbf{A}_{EQ} = \rho(\mathbf{A}_1) = \overline{\mathbf{A}_1}$ (e.g. $\overline{pop}(\overline{push}(x, X)) = X$, etc.); then, correctness can be directly checked at the constructive level.

7 Correctness proofs

Of course, we cannot accept an implementation which does not completely simulate "from the user point of view" the abstract data structure described by $\mathbf{SPEC}_1$. From the above semantics, this means that an abstract implementation must (at least) satisfy the following criteria:

- Each operation to be implemented ($\in \Sigma_1$) has a complete constructive representation (in the "product values" synthesized by $\mathbf{\Sigma}_{SYNTH}$).
- The *user view* of **IMPL** is isomorphic to the descriptive view associated with $\mathbf{SPEC}_1$. This means that SEM_{IMPL} must be isomorphic to $T_{\mathbf{SPEC}_1}$.

These two criteria are handled in four steps (by dividing the second one into three conditions):

- The complete implementation of all operations to be implemented is called *operation-completeness*.
- SEM_{IMPL} must be finitely generated over Σ_1. This means that SEM_{IMPL} is an object of the subcategory $\text{Gen}(\mathbf{S}_1, \Sigma_1, \emptyset)$ of $\text{Alg}(\mathbf{S}_1, \Sigma_1, \emptyset)$. This condition is called *data protection*.
- SEM_{IMPL} must be a $\mathbf{SPEC}_1$-algebra. This means that SEM_{IMPL} must validate the $\mathbf{SPEC}_1$-axioms ($\mathbf{A}_1$). This condition is called the *validity* of **IMPL**.
- Finally, among all finitely generated $\mathbf{SPEC}_1$-algebras, SEM_{IMPL} must be initial. This condition is called the *consistency* of **IMPL**. Now, SEM_{IMPL} is necessarily isomorphic to $T_{\mathbf{SPEC}_1}$ (unicity of the initial object).

An abstract implementation satisfying these four conditions is called *acceptable*. Some other correctness criteria will be added. For instance, acceptability only concerns the objects to be implemented; it does not ensure the protection of resident values.

Notice that the last acceptability condition reflects an initial view of abstract data types. Of course, the consistency condition can be modified according to a loose semantics "protecting some predefined specifications" [SW83], [Ber87] by simply requiring consistency of SEM_{IMPL} with respect to these predefined specifications.

7.1 Operation completeness

Operation completeness was first introduced by [EKP80]. The fact that all operations to be implemented have a synthesized constructive representation means that all Σ_1-terms have a synthesized constructive representation.

Definition 14 : **IMPL** is *op-complete* if and only if for all terms $t \in T_{\Sigma_1}$, there is $\alpha \in T_{SORTimpl}$ such that $\overline{\rho_s}(t) = \alpha$ in T_{REP} (where s is the sort of t).

Notice that the operation-implementing axioms ($\mathbf{A}_{OP}$) must entirely (recursively) define the implementation of all operations. This must be done without any consideration of the equality representation (i.e. without using $\mathbf{A}_{EQ}$). For example, given a representation $< t, 1 >$ of the term $push(n, empty)$, we must be able to directly apply pop : $\overline{pop}(< t, 1 >) =< t, 0 >$, without looking at the implementation of the term *empty*. Thus, op-completeness is defined in T_{REP} and not in T_{EQ}.

The following theorem shows that op-completeness can always be checked at the constructive level (i.e. without explicitly using the representation).

Theorem 15 : **IMPL** is *op-complete* if and only if for all terms $\overline{t} \in T_{\overline{\Sigma_1}}$, there is $\alpha \in T_{SORTimpl}$ such that $\overline{t} = \alpha$ in T_{OPimpl} .

Proof: From the specification of $\mathbf{A}_{REP}$, for each Σ_1-term t, $\overline{\rho_s}(t)$ is equal to the $\overline{\Sigma_1}$-term $\overline{t} = \rho(t)$. Consequently, if all $\overline{\Sigma_1}$-terms have a synthesized value for **OPimpl**, then a fortiori all Σ_1-terms have a synthesized representation for **REP**. Conversely, ρ is a

surjective signature morphism, and **REP** is consistent over **OPimpl**. Thus, if for each Σ_1-term t, the term $\overline{t} = \overline{\rho_s}(t)$ is equal to a synthesized value α in T_{REP} , then each $\overline{\Sigma_1}$-term $\overline{t}$ is equal to a synthesized value α in T_{OPimpl} . □

Consequently, op-completeness is not difficult to check. It can be directly proved by structural induction over $\overline{\Sigma_1}$. Moreover, we have the following result:

Corollary 16 : If **OPimpl** is sufficiently complete over **SORTimpl**, then **IMPL** is op-complete.

Proof: Immediate, because **OPimpl** adds $\overline{\Sigma_1}$ to **SORTimpl** and sufficient completeness means that the canonical adjunction morphism from $T_{SORTimpl}$ to T_{OPimpl} is surjective. □

Sufficient completeness of **OPimpl** over **SORTimpl** is not needed in the general case. For instance, we may think of a SET by $STRING$ implementation where $\overline{del}(\langle "aaa" \rangle_{SET})$ does not return any string. Then, **OPimpl** will not be sufficiently complete over **SORTimpl**. Nevertheless, since $\langle "aaa" \rangle_{SET}$ is not a reachable value, this fact does not destroy op-completeness of **IMPL**. We only need for $\overline{del}$ to return a string when its argument is non redundant. However, this corollary works in most examples. For instance, $\overline{del}(\langle "aaa" \rangle_{SET}) = \langle "aa" \rangle_{SET}$ in Example 11. Similar results were first given in [EKP80].

Example 17 : We prove that our implementation of $STACK$ by $ARRAY$ is op-complete, by structural induction.

- $\overline{\rho_{STACK}}(empty)$ is equal to $\overline{empty}$, which is equal to $\alpha = \langle create, 0 \rangle_{STACK}$
- if x and X have constructive representations ($x = \alpha_1 = \langle n \rangle_{NAT}$; and $\overline{\rho_{STACK}}(X) = \alpha_2 = \langle t, i \rangle_{STACK}$), then so does $\overline{\rho_{STACK}}(push(x, X))$:

$$\overline{\rho_{STACK}}(push(x, X)) = \overline{push}(\langle n \rangle_{NAT}, \langle t, i \rangle_{STACK})$$

$$\overline{push}(\langle n \rangle_{NAT}, \langle t, i \rangle_{STACK}) = \langle t[i] := n \ , \ succ(i) \rangle_{STACK} = \alpha_3$$

- similar reasonings apply for *pop* and *top*.

7.2 Data protection

Definition 18 : **IMPL** is *data protected* if and only if the semantical result SEM_{IMPL} is finitely generated over Σ_1 .

Theorem 19 : If **H** is sufficiently complete over **SP**, then **IMPL** is data protected. (**SP**=(**S**,Σ,**A**) is the common specification between **SPEC**$_0$ and **SPEC**$_1$.)

Proof: The specification of an abstract implementation does not contain any operation with target sort in $\mathbf{S}_1 - \mathbf{S}$, except those of Σ_1. Thus, it suffices to prove that SEM_{IMPL} is finitely generated with respect to the sorts of **S**. Since SEM_{IMPL} is included in T_{EQ}, it suffices to prove that T_{EQ} is finitely generated with respect to T_{SP} ; i.e.

that $\mathbf{EQ}+\mathbf{REP}+..+\mathbf{SPEC}_0$ is sufficiently complete over $\mathbf{SP}$. Consequently, Theorem 19 results from the fact that the abstract implementation specification does not contain any operation with target sort in $\mathbf{S}$, except those of Σ_1 and $\mathbf{\Sigma}_H$. □

From the theoretical point of view, sufficient completeness of the hidden component is not required, since $\mathbf{A}_{OP}$ or $\mathbf{A}_{EQ}$ may complete the specification of some hidden operations. However it is clearly suitable from a methodological point of view, as $\mathbf{A}_{EQ}$ and $\mathbf{A}_{OP}$ have not to play this role.

Data protection is not difficult to prove, since it can be proved by structural induction or via syntactical tools (such as *fair presentations*, [Bid82]). Our $STACK$ by $ARRAY$ example is clearly data protected, as $\mathbf{H}$ is empty. Example 11 (SET by $STRING$) is also data protected because *remove* and *occurs* always return predefined strings or booleans ($\mathbf{A}_H$ is equivalent to a canonical rewriting system).

7.3 Validity

Definition 20 : **IMPL** is a *valid* abstract implementation if and only if for all Σ_1-terms, t and t', we have:

$$\text{if } t = t' \text{ in } T_{\mathbf{SPEC}_1} \text{ then } t = t' \text{ in } SEM_{IMPL}.$$

The following results prove that validity is equivalent to the fact that SEM_{IMPL} validates $\mathbf{SPEC}_1$; they also prove that validity can always be reduced to a hierarchical consistency property.

Theorem 21 : If **IMPL** is data protected then the following conditions are equivalent:

1. **IMPL** is a valid abstract implementation
2. there is a Σ_1-morphism from $T_{\mathbf{SPEC}_1}$ to SEM_{IMPL}
3. SEM_{IMPL} validates the axioms of $\mathbf{A}_1$
4. SEM_{IMPL} validates the axioms of $\mathbf{A}_1 - \mathbf{A}$
5. T_{EQ} validates the axioms of $\mathbf{A}_1 - \mathbf{A}$
6. **ID** is hierarchically consistent over $\mathbf{EQ}+\mathbf{REP}+..+\mathbf{SPEC}_0$

where **ID** is the presentation over $\mathbf{EQ}+..+\mathbf{SPEC}_0$ which contains the set of axioms $\mathbf{A}_1 - \mathbf{A}$. Thus, $\mathbf{ID}+\mathbf{EQ}+..+\mathbf{SPEC}_0$ contains all the specifications involved in our formalism (both the specification associated with **IMPL** and the descriptive specification $\mathbf{SPEC}_1$).

Proof: [$1 \Longleftrightarrow 2$] is clear : since $T_{\mathbf{SPEC}_1}$ is finitely generated over Σ_1, there is a morphism from $T_{\mathbf{SPEC}_1}$ to SEM_{IMPL} if and only if two Σ_1-terms equal in $T_{\mathbf{SPEC}_1}$ are also equal in SEM_{IMPL}.

[$2 \Longleftrightarrow 3$] results from the facts that SEM_{IMPL} is finitely generated over Σ_1 and that $T_{\mathbf{SPEC}_1}$ is initial in $\mathbf{SPEC}_1$. Thus, there is a morphism from $T_{\mathbf{SPEC}_1}$ to SEM_{IMPL} if and only if SEM_{IMPL} is a $\mathbf{SPEC}_1$-algebra (i.e. SEM_{IMPL} validates $\mathbf{A}_1$).

$[3 \Longleftrightarrow 4]$ results from the fact that **EQ+..+SPEC**$_0$ contains **SPEC**$_0$. In particular, it contains **SP**, thus it contains **A**. Consequently, SEM_{IMPL} always validates **A**.

$[4 \Longleftrightarrow 5]$ results from the facts that the axioms of $\mathbf{A}_1 - \mathbf{A}$ only concern the signature $(\mathbf{S}_1, \Sigma_1)$, and $SEM_{IMPL} = U_{\Sigma_1}(T_{EQ})$.

$[5 \Longleftrightarrow 6]$ results from the fact that **ID** does not add new operations to **EQ+..+SPEC**$_0$ (**ID**$=\mathbf{A}_1 - \mathbf{A}$). Thus, **ID** is hierarchically consistent over **EQ+..+SPEC**$_0$ if and only if T_{EQ} already validates the axioms of $\mathbf{A}_1 - \mathbf{A}$. □

The main result is the equivalence between the validity of **IMPL** and the consistency of **ID** over **EQ+..+SPEC**$_0$. Thus, validity proofs can always be handled by "classical" methods. This feature is entirely due to our intermediate constructive sorts and the equality representation explicitly specified via $\mathbf{A}_{EQ}$.

Example 22 : The validity of our abstract implementation of $STACK$ is shown by proving that each $STACK$-axiom is a theorem of the specification associated with **IMPL**. We prove here that $pop(push(n, X))$ is equal to X in T_{EQ}. The other axioms of $STACK$ are proved in a straightforward manner, following the same method.

Since $\mathbf{A}_{REP}$ contains the axiom

$$\overline{\rho_{STACK}}(X) = \overline{\rho_{STACK}}(Y) \Longrightarrow X = Y$$

and since our implementation is op-complete, it suffices to show that the term $\overline{pop}(\overline{push}(\langle n \rangle_{NAT}, \langle t, i \rangle_{STACK}))$ is equal to $\langle t, i \rangle_{STACK}$ in T_{EQ} . From $\mathbf{A}_{OP}$, it results that

$$\overline{pop}(\overline{push}(\langle n \rangle_{NAT}, \langle t, i \rangle_{STACK})) = \langle t[i] := n, i \rangle_{STACK} \ .$$

Moreover, from the equality representation (specified in $\mathbf{A}_{EQ}$), it results that

$$Astackt[i] := ni = \langle t, i \rangle_{STACK}$$

which ends our proof.

To prove that our implementation of SET by $STRING$ is valid, it suffices to prove that each axiom of SET is true in T_{EQ}. We will prove here that

$$ins(x, ins(y, X)) = ins(y, ins(x, X)) \text{ is true in } T_{EQ}.$$

Since $\mathbf{A}_{REP}$ contains the following axiom:

$$\overline{\rho_{SET}}(X) = \overline{\rho_{SET}}(Y) \implies X = Y$$

and since our implementation is op-complete, it suffices to prove that

$\overline{ins}(\langle n \rangle_{NAT}, \overline{ins}(\langle m \rangle_{NAT}, \langle s \rangle_{SET})) = \overline{ins}(\langle m \rangle_{NAT}, \overline{ins}(\langle n \rangle_{NAT}, \langle s \rangle_{SET}))$

We have to distinguish 5 cases:

- n and m both occur in s ; then we get

$$\langle s \rangle_{SET} =?= \langle s \rangle_{SET}$$

- n occurs in s and m does not; then we get

$$\langle add(m,s)\rangle_{SET} =?= \langle add(m,s)\rangle_{SET}$$

- m occurs in s and n does not; then we get

$$\langle add(n,s)\rangle_{SET} =?= \langle add(n,s)\rangle_{SET}$$

- s does not contain n and m, but n and m are equals; then we get

$$\langle add(n,s)\rangle_{SET} =?= \langle add(m,s)\rangle_{SET} \qquad \text{(with } n=m\text{)}$$

- s does not contain n and m, and n and m are distinct; then we get

$$\langle add(n, add(m,s))\rangle_{SET} =?= \langle add(m, add(n,s))\rangle_{SET}$$

The four first equalities are trivial. The last one results from the equality representation.

7.4 Consistency

Definition 23 : **IMPL** is *consistent* if and only if for all Σ_1-terms, t and t', we have:

$$\text{if } t=t' \text{ in } SEM_{IMPL}, \text{ then } t=t' \text{ in } T_{\mathbf{SPEC}_1}.$$

The following results prove that consistency is equivalent to the fact that SEM_{IMPL} is initial in Gen($\mathbf{SPEC}_1$). They also prove that consistency can always be reduced to a hierarchical consistency property.

Theorem 24 : If **IMPL** is data protected and valid, then the following conditions are equivalent:

1. for all t and t' in T_{Σ_1}, if $t=t'$ in T_{EQ} then $t=t'$ in $T_{\mathbf{SPEC}_1}$
2. **IMPL** is consistent
3. the initial morphism from $T_{\mathbf{SPEC}_1}$ to SEM_{IMPL} is a monomorphism
4. SEM_{IMPL} is an initial $\mathbf{SPEC}_1$-algebra
5. the initial morphism from $T_{\mathbf{SPEC}_1}$ to $U_{\Sigma_1}(T_{ID})$ is a monomorphism
6. **ID+EQ+..+SPEC**$_0$ is hierarchically consistent over $\mathbf{SPEC}_1$

Proof: [1 $\Longleftrightarrow$ 2] results from the fact that SEM_{IMPL} is equal to the part of T_{EQ} concerning the signature $(\mathbf{S}_1, \Sigma_1)$.

[2 $\Longleftrightarrow$ 3] results from the fact that $T_{\mathbf{SPEC}_1}$ is finitely generated over Σ_1. Notice that the initial morphism $T_{\mathbf{SPEC}_1} \to SEM_{IMPL}$ exists, from Theorem 21.

[3 $\Longleftrightarrow$ 4] results from the fact that SEM_{IMPL} is finitely generated over Σ_1.

[3 $\Longleftrightarrow$ 5] results from $SEM_{IMPL} = U_{\Sigma_1}(T_{EQ})$, and from $T_{EQ} = T_{ID}$ (Theorem 21).

$[5 \Longleftrightarrow 6]$ is clear since the initial morphism $T_{\mathbf{SPEC}_1} \to U_{\Sigma_1}(T_{ID})$ is the adjunction unit associated with the presentation **ID+..+SPEC**$_0$ over **SPEC**$_1$. □

For the same reasons as Theorem 21, Theorem 24 facilitates the consistency proofs, since they can be handled using rewriting techniques or structural induction.

Example 25 : The only axioms that may destroy the consistency of **ID+..+SPEC**$_0$ over **SPEC**$_1$ are the axioms of sort in $\mathbf{S}_1$. In the *STACK* by *ARRAY* example, these axioms are:

$$\overline{\rho_{STACK}}(X) = \overline{\rho_{STACK}}(Y) \implies X = Y$$

$$\overline{\rho_{NAT}}(m) = \overline{\rho_{NAT}}(n) \implies m = n$$

These axioms imply to prove that two descriptive terms represented by the same constructive value (in T_{EQ}) are equal (in $T_{\mathbf{SPEC}_1}$). Thus, we must consider each axiom of $\mathbf{A}_{OP} \cup \mathbf{A}_{EQ}$, and prove that it does not create inconsistencies. Let us consider, for instance, the axiom:

$$\overline{push}(\langle n \rangle_{NAT}, \langle t, i \rangle_{STACK}) \ = \ \langle t[i] := n, succ(i) \rangle_{STACK} \ .$$

Since we work on the stack *values* (not on the stack ground terms), we can handle our proofs with respect to the normal forms of $STACK$. It is possible to prove, by structural induction, that $\langle t, i \rangle_{STACK}$ represents the stack

$$push(t[i\text{-}1], push(\cdots, push(t[0], empty)..))$$

Thus, our proof is clear, as $\overline{push}(\overline{\rho_{NAT}}(n), \overline{\rho_{STACK}}(X))$ represents $push(n, X)$. Other axioms are handled in a similar manner using the normal forms.

In the SET by $STRING$ example, axioms whose sort belongs to $\mathbf{S}_1$ are:

$$\overline{\rho_{SET}}(X) = \overline{\rho_{SET}}(Y) \implies X = Y$$

$$\overline{\rho_{NAT}}(n) = \overline{\rho_{NAT}}(m) \implies n = m$$

$$\overline{\rho_{BOOL}}(a) = \overline{\rho_{BOOL}}(b) \implies a = b$$

These axioms imply to show that two descriptive terms represented by the same constructive value (in T_{EQ}), are equal (in $T_{\mathbf{SPEC}_1}$). Thus, we must consider each axiom of $\mathbf{A}_H \cup \mathbf{A}_{OP} \cup \mathbf{A}_{EQ}$, and prove that it does not create inconsistencies. The consistency of $\mathbf{A}_H$ is not difficult to prove. Before proving the consistency of the other axioms, we first prove the following "lemma:" if the string s represents the set X then

$$x \in X = occurs(x, s)$$

(this results from the last axiom of $\mathbf{A}_{OP}$ in Example 11 and from the axiom $\langle n \rangle_{NAT} = \overline{\rho_{NAT}}(n)$ of $\mathbf{A}_{REP}$, since NAT is the common specification between SET and $STRING$).

Then, similar to the use of normal forms in the $STACK$ example, we remark that $\emptyset$ is represented by λ (first axiom of $\mathbf{A}_{OP}$); and if s represents X then $ins(n, X)$ is represented by $add(n, s)$ each time $n \in X$ is false. (This results from the second axiom of $\mathbf{A}_{OP}$ and from our "lemma.")

Next, the consistency of all axioms of **IMPL** is straightforward. For example, from the following axioms

$$occurs(x,s) = False \Longrightarrow \overline{ins}(\langle x \rangle_{NAT}, \langle s \rangle_{SET}) = \langle add(x,s) \rangle_{SET}$$

$$occurs(x,s) = True \Longrightarrow \overline{ins}(\langle x \rangle_{NAT}, \langle s \rangle_{SET}) = \langle s \rangle_{SET}$$

we get:

$$x \in X = False \implies ins(x,X) = ins(x,X)$$

$$x \in X = True \implies ins(x,X) = X$$

which do not create inconsistencies.

From the equality representation

$$\langle add(x, add(y,s)) \rangle_{SET} = \langle add(y, add(x,s)) \rangle_{SET}$$

we get:

$$ins(x, ins(y,X)) = ins(y, ins(x,X))$$

which does not create inconsistencies.

Remark 26 : Theorems 3 and 4 reduce the most difficult correctness proofs to *hierarchical consistency* criteria, which mainly leads to theorem proving methods. It is well known that, in many cases, hierarchical consistency is difficult to check. Nevertheless, these results focalize the implementation correctness problem to this well known abstract data type problem. Moreover, hierarchical consistency is considerably more usable than purely semantical criteria, such as the existence of a morphism. Also, it should be noted that the semantical reasonings introduced in Example 25 can be replaced by more systematic (but less concise) methods based on rewriting theory.

Now, we are able to define the acceptability of an abstract implementation:

Definition 27 : **IMPL** is *acceptable* if and only if it is op-complete, data protected, valid and consistent.

7.5 Correct implementations

When defining *acceptability* of abstract implementation so far, we were only interested in the implemented data structure (initially described by $\mathbf{SPEC}_1$). Most existing abstract implementation formalisms do not add other conditions for correctness. However, acceptability do not care about the interactions between the implementation and other specifications (used by, or using, the implementation). In particular, acceptable implementations may alter already implemented (resident) specifications.

Definition 28 : **(protection of the resident specification).** An implementation **IMPL** *protects the resident data structure* if and only if $\mathbf{EQ}$+..+$\mathbf{SPEC}_0$ is persistent over the resident specification $\mathbf{SPEC}_0$.

Protection of the resident data structure implies that the semantical result SEM_{IMPL} is finitely generated over Σ_1 :

Proposition 29 : Let **IMPL** be any abstract implementation. If **IMPL** protects the resident data structure then **IMPL** is data protected (definition 18).

Proof: In the proof of Theorem 19, we showed that it suffices to prove that **EQ+..+SPEC**$_0$ is sufficiently complete over **SP**. By hypothesis, **SPEC**$_0$ contains **SP**, and **SPEC**$_0$ is sufficiently complete over **SP**. Thus, the sufficient completeness of **EQ+..+SPEC**$_0$ over **SPEC**$_0$ gives the conclusion. □

As already mentioned in the beginning of section 7, our acceptability criteria are related to an initial semantics. In particular, the validity of **IMPL** signifies that the initial algebra T_{EQ} validates the **SPEC**$_1$ axioms (Theorem 21). It implies that all *finitely generated* (**EQ+..+SPEC**$_0$)-algebras validate $\mathbf{A}_1$ but it does not implies that all (**EQ+..+SPEC**$_0$)-algebras validate $\mathbf{A}_1$ (see Example 34 below).

Definition 30 : (Full validity). An implementation is *fully valide* if and only if all (**EQ+..+SPEC**$_0$)-algebras validate $\mathbf{A}_1$.

Notice that, in practice, this exactly means that the *equality representation* is powerful enough; in such a way that validity can be proved by equational reasoning, without using structural induction. In particular, all implementations following the [EKMP82] semantics (i.e. $\mathbf{A}_{EQ} = \mathbf{A}_1$ in our framework, cf. Section 6) are trivialy fully valide.

Similarly to Theorem 21, an implementation is fully valide if and only if for each (**EQ+..+SPEC**$_0$)-algebra A the adjunction morphism from A to $F_{ID}(A)$ is injective (i.e. if and only if **ID** is *strongly hierarchically consistent* over **EQ+..+SPEC**$_0$).

Definition 31 : (Correct implementations). An abstract implementation is *correct* if and only if it is op-complete, it protects the resident data structure, it is fully valide and it is consistent.

Of course, “correct” implies “acceptable.”

8 Reuse of abstract implementations

8.1 Implementations and enrichments

Let **SPEC**$_1$ be a specification implemented via **IMPL**. Let **PRES** be a presentation over **SPEC**$_1$. We have shown (Section 2.3) that every proof concerning **PRES** is done with respect to **SPEC**$_1$, but not with respect to the specification of **IMPL**. The constructive implementation of **PRES+SPEC**$_1$ is not specified by **PRES+SPEC**$_1$, it is specified by **PRES+EQ+..+SPEC**$_0$, where **EQ+..+SPEC**$_0$ is the whole specification of the implementation of **SPEC**$_1$. The following theorem proves that everything is going well whenever the presentation **PRES** is persistent over **SPEC**$_1$: the “user view” of the constructive specification **PRES+EQ+..SPEC**$_0$ is isomorphic to the descriptive data

structure specified by **PRES+SPEC$_1$**. This result is entirely due to our intermediate constructive sorts.

Theorem 32 : If **IMPL** is a correct abstract implementation of **SPEC$_1$**, then for all persistent presentations **PRES** over **SPEC$_1$**, we have:

$$U_{\Sigma_1+\mathbf{\Sigma}_{PRES}}(T_{EQ+PRES}) \;=\; T_{SPEC_1+PRES}$$

This theorem proves that the presentation **PRES**, together with the abstract implementation of **SPEC$_1$**, always provides the user with the expected results.

Before proving Theorem 32, we recall the following lemma (proved in [Ber86] with positive conditional axioms).

Lemma 33 : If $\mathbf{P}_a$ and $\mathbf{P}_b$ are two *persistent* presentations over a specification **Spec**, with disjoint signatures, then $\mathbf{P}_b$ is still a persistent presentation over ($\mathbf{P}_a$+**Spec**).

Proof: (of Theorem 32) We remark that correctness of **IMPL** ensure that **ID+..+SPEC$_0$** is persistent over **SPEC$_1$** (Theorem 24 and data protection). Thus, we deduce from the previous lemma that **PRES+ID+..+SPEC$_0$** is persistent over **SPEC$_1$+PRES** : $U_{\Sigma_1+\mathbf{\Sigma}_{PRES}}(T_{ID+PRES})$ is isomorphic to T_{SPEC_1+PRES} . Consequently, it suffices to prove that $T_{ID+PRES}$ is isomorphic to $T_{EQ+PRES}$. This results from the fact that all (**EQ+..+SPEC$_0$**)-algebras validate $\mathbf{A}_1$ (full validity). □

Here is an example of *acceptable* implementation which is not *correct*, and an example of persistent presentation which does not cope with this implementation.

Example 34 : Let **SPEC$_0$** defined by $\mathbf{S}_0 = \{NAT\}$, $\Sigma_0 = \{0,\ succ_\}$ with usual arities, and $\mathbf{A}_0 = \emptyset$. Let **SPEC$_1$** defined by

$$\mathbf{S}_1 = \{UNAT\}$$

$$\Sigma_1 = \{zero :\rightarrow UNAT\ ,\ next :\ UNAT \rightarrow UNAT\}$$

and $\mathbf{A}_1$ containing the following axiom:

$$next(x) = next(y) \Longrightarrow x = y$$

Of course the initial algebras are isomorphic (to $I\!N$), thus $UNAT$ can be implemented by NAT with $\mathbf{A}_H = \mathbf{A}_{EQ} = \emptyset$. The implementation is clearly acceptable; however it is not correct (not fully valide).

Let **PRES** simply adding a constant τ to the signature, and the axiom:

$$next(\tau) = next(zero)$$

PRES is persistent over **SPEC$_1$** $= UNAT$ (τ must be equal to $zero$) but it is not persistent over the implementation of **SPEC$_1$** (τ is not equal to $zero$ in $T_{PRES+EQ+..+NAT}$: the implementation does not ensure the injectivity of $next$). $T_{PRES+EQ+..+NAT}$ is a non finitely generated (**EQ+..+**NAT)-algebra which does not validate the axiom of $\mathbf{A}_1$.

8.2 Composition of abstract implementations

When we implement $\mathbf{SPEC}_1$ by means of $\mathbf{SPEC}_0$, the resident specification $\mathbf{SPEC}_0$ is often already implemented by means of a lower level implementation. But all our correctness proofs are done with respect to the descriptive specification $\mathbf{SPEC}_0$, not with respect to the specification of the implementation of $\mathbf{SPEC}_0$. We prove in this section that the composition of two correct implementations always yields correct results. This feature is not provided in any work already put forward. The formalism of [SW82] provides correct "vertical compositions," but these vertical compositions do not solve our problem: all upper level implementation operations must be implemented by the lower level implementation. This results in a large number of operations being implemented by the lowest level implementation; moreover, this implies that all the lower level implementations must be redefined every time a new implementation is added.

The following theorem proves that the user view, obtained after pushing two correct abstract implementations together, is always correct.

Theorem 35 : Let **IMPL_2** be an abstract implementation of **SPEC_2** by means of $\mathbf{SPEC}_1$. Let **IMPL_1** be an abstract implementation of $\mathbf{SPEC}_1$ by means of $\mathbf{SPEC}_0$. Consider the specification **IMPL**(1,2) obtained from the specification of **IMPL_2** by substituting the specification of **IMPL_1** for $\mathbf{SPEC}_1$. Thus **IMPL**(1,2) is equal to the specification

$\mathbf{SPEC}_0$+(**SORTimpl_1**+$\mathbf{H}_1$+⋯+**EQ_1**)+(**SORTimpl_2**+$\mathbf{H}_2$+⋯+**EQ_2**)

If **IMPL_1** and **IMPL_2** are both correct, then we have:

$$U_{\Sigma_2}(T_{IMPL(1,2)}) = T_{SPEC_2}$$

Proof: Since **IMPL_2** is correct, (**SORTimpl_2** + ⋯ + **EQ_2**) is persistent over $\mathbf{SPEC}_1$. Thus, Theorem 5 (with **PRES**=**SORTimpl_2** + ⋯ + **EQ_2**) proves that $U_{\Sigma(SPEC_1+..+EQ_2)}(T_{IMPL(1,2)}) = T_{EQ_2}$. In particular, $U_{\Sigma_2}(T_{IMPL(1,2)}) = U_{\Sigma_2}(T_{EQ_2}) = SEM_{IMPL_2}$. Moreover, the correctness of **IMPL_2** implies that $SEM_{IMPL_2} = T_{SPEC_2}$, which ends our proof. □

This theorem can be extended to a finite number of correct implementations. Thus, it is possible to handle structured, modular abstract implementations. This provides a formal foundation for a methodology of program development by stepwise refinement.

9 Conclusion

The abstract implementation formalism described in this paper relies on three main ideas:

- Abstract implementation is done by means of intermediate *constructive* values, which are distinct from the *descriptive* values to be implemented.

- These constructive sorts are synthesized by means of *synthesis operations* which extend the classical notion of abstraction operations.

  The correspondence between the descriptive sorts/operations to be implemented and the constructive sorts/operations is specified by means of a *representation signature isomorphism*.
- The *equality representation* is explicitly introduced into the abstract implementation in order to handle conditional axioms.

The main results of this abstract implementation formalism are the following:

- It allows use of *positive conditional axioms*, which facilitate the specifications.
- All correctness proof criteria for abstract implementation are "simple" ones (sufficient completeness, hierarchical consistency or fair presentations). This feature provides the specifier with "classical" methods such as *term rewriting* methods, *structural induction* methods or *syntactical* criteria.
- Abstract implementations are compatible with the notion of *enrichment*.
- The composition of several correct implementations always yields correct results. Thus, abstract implementations can be specified in a modular and structured way.

As a last remark, we want to emphasize the fact that the semantical level of our abstract implementation is built with "simple" functors. Consequently, it is not difficult to extend this formalism, for instance to abstract data types with *exception handling* [BBC86], [Ber86], or to *parameterization* since parameterization mainly relies on synthesis functors and pushouts (see [ADJ80]).

Acknowledgements : It is a pleasure to express gratitude to Michel Bidoit, Christine Choppy and Marie-Claude Gaudel for interesting suggestions, stimulating discussions and careful reading of previous versions of this paper. This work was partially supported by ESPRIT Project METEOR, at LRI (Orsay, France).

References

[ADJ76] Goguen J., Thatcher J., Wagner E. : "*An initial algebra approach to the specification, correctness, and implementation of abstract data types*", Current Trends in Programming Methodology, Vol.4, Yeh Ed. Prentice Hall, 1978 (also IBM Report RC 6487, October 1976).

[ADJ78] Goguen J.A., Thatcher J.W., Wagner E.G. : "*Abstract data types as initial algebras and the correctness of data representation*", Current Trends in Programming Methodology 4, (Yeh R. Ed), Prentice-Hall, 1978, p.80-149.

[ADJ80] Ehrig H., Kreowski H., Thatcher J., Wagner J., Wright J. : "*Parameterized data types in algebraic specification langages*", Proc. 7th ICALP, July 1980.

[BBC86] Bernot G., Bidoit M., Choppy C. : "*Abstract data types with exception handling : an initial approach based on a distinction between exceptions and errors*", Theoretical Computer Science, Vol.46, No.1, p.13-45, November 1986.

[BCFG86] Bouge' L., Choquet N., Fribourg L., Gaudel M-C. : "*Test sets generation from algebraic specifications using logic programming*", Journal of Systems and Software, vol.6, No.4, November 1986.

[Ber86] Bernot G. : "*Une se'mantique alge'brique pour une spe'cification diffe'rencie'e des exceptions et des erreurs : application a' l'imple'mentation et aux primitives de structuration des spe'cifications formelles*", The'se de troisie'me cycle, Universite' de Paris-Sud, Orsay, February 1986.

[Ber87] Bernot G. : "*Good functors... are those preserving philosophy !*", Proc. Summer Conference on Category Theory and Computer Science, September 1987, Springer-Verlag LNCS 283, p.182-195.

[Bid82] Bidoit M. : "*Algebraic data types: structured specifications and fair presentations*", Proc. of AFCET Symposium on Mathematics for Computer Science, Paris, March 1982.

[Bou82] Bouge' L. : "*Mode'lisation de la notion de tests de programme*", The'se de troisie'me cycle, Universite' de Paris VI, October 1982.

[EKMP82] Ehrig H., Kreowski H., Mahr B., Padawitz P. : "*Algebraic implementation of abstract data types*", Theoretical Computer Science 20, 1982, p.209-263.

[EKP80] Ehrig H., Kreowski H., Padawitz P. : "*Algebraic implementation of abstract data types: concept, syntax, semantics and correctness*", Proc. ICALP, Springer-Verlag LNCS 85, 1980.

[Gau80] Gaudel M.C. : "*Ge'ne'ration et preuve de compilateurs base'e sur une se'mantique formelle des langages de programmation*", The'se d'e'tat, Nancy, 1980.

[Gau86] Gaudel M-C. : "*Automation of testing in software development*", Position paper, IFIP 86, panel on Automation of software development, Dublin (Ireland), September 1986.

[GHM76] Guttag J.V., Horowitz E., Musser D.R. : "*Abstract data types and software validation*", C.A.C.M., Vol 21, n.12, 1978. (also USG ISI Report 76-48).

[GM88] Gaudel M-C., Moineau T. : "*A theory of software reusability*", to appear in Proc. ESOP 88, Nancy, Mars 1988.

[Gut75] Guttag J.V. : "*The specification and application to programming*", Ph.D. Thesis, University of Toronto, 1975.

[Hoa72] Hoare C.A.R. : "*Proofs of correctness of data representation*", Acta Informatica 1, No.1, 1972, p.271-281.

[LZ75] Liskov B., Zilles S. : "*Specification techniques for data abstractions*", IEEE Transactions on Software Engineering, Vol.SE-1 N 1, March 1975.

[San87] Sannella D. : "*Implementations revisited*", 5th Workshop on Specification of abstract data types, Edinburgh, September 1987, abstracts in LFCS report 87-41.

[Sch87] Schoett O. : "*Data abstraction and the Correctness of Modular Programming*", PhD thesis, University of Edinburgh, 1987.

[SW82] Sannella D., Wirsing M. : "*Implementation of parameterized specifications*", Report CSR-103-82, Department of Computer Science, University of Edinburgh.

[SW83] Sannella D., Wirsing M. : "*A kernel language for algebraic specification and implementation*", Proc. Intl. Conf. on Foundations of computation Theory, Springer-Verlag, LNCS 158, 1983.

Chapitre 3 :
Testing against formal specifications : a theoretical view

Gilles BERNOT

LIENS, CNRS URA 1327
Ecole Normale Supérieure,
45 Rue d'Ulm,
F-75230 PARIS CEDEX 05
FRANCE

bitnet : `bernot@frulm63`
uucp : `bernot@ens.ens.fr`

(Appeared in TAPSOFT CCPSD, Brighton U.K., April 1991, Springer-Verlag LNCS 494, p.99-119.)

Table des matières

1 Introduction 76

2 Intuitive approach 77

3 Notations 79

4 Testing contexts 80

5 The refinement preorder 83

6 Case of algebraic specifications 84

7 Regularity and uniformity 86

8 The oracle 88

9 Partial oracles 89

10 Recapitulation 91

Résumé

Cet article développe une théorie pour le test de logiciel. Si l'on dispose d'une *spécification formelle* des propriétés attendues du logiciel, on peut étudier formellement la validation de logiciel par rapport à sa specification. L'avantage principal de la théorie développée ici est de préciser explicitement ce que l'on fait lorsque le processus de validation mélange des méthodes de preuve (pour les propriétés cruciales) et des techniques de test (pour les autres propriétés). Deux étapes sont essentielles pour le test : le choix d'un jeu de test, puis la décision du succès ou de l'échec de ce jeu de test lorsqu'il est soumis au programme. Nous montrons qu'il est possible de déterminer la qualité d'un jeu de test, ainsi que la fiabilité de la décision de succès ou échec, par le biais d'*hypothèses* sur le programme testé. Nous décrivons alors des schémas d'hypothèse qui permettent de sélectionner automatiquement des jeux de test à partir d'une specification algébrique.

1 Introduction

Assuming that a formal specification is available, one can formally study the validation of a software with respect to its specification. While proof theories are widely investigated, testing theories have not been extensively studied. The idea of deriving test data sets from a specification can be found in [Rig85], [Scu88], but there are few other published works. The work reported in this paper is the continuation of the works about formal specifications and testing reported in [Bou85], [BCFG85], [BCFG86] and more recently [GM88]. Some pioneering works on that subject were [GHM81] and [GCG85].

In practice, when big softwares are involved, a complete proof is often impossible, or at least it is not realistic because it would be too costly. The crucial properties of the program under test should be proved, but several less critical properties can be checked by testing. The aim of this paper is to propose a formal model of the testing approach. We assume that a formal specification (i.e. an axiomatic specification) is given and we have to validate a program against its specification: some of the axioms can be proved while the other ones can be tested.

Most of the current methods and tools for software testing are based on the structure of the program to be tested ("white-box" testing). Using a formal specification, it becomes possible to also start from the specification to define some testing strategies in a rigorous and formal framework. These strategies provide a formalization of the well known "black-box" testing approaches. They have the interesting property to be independent of the program; thus, they result in test data sets which remain unchanged even when the program is modified. Moreover, such strategies allow to test if all cases mentioned in the specification are actually dealt with in the program.

The main advantage of our approach is to formally express what we do when testing. It allows also to modelize cases where some properties have been proved. Moreover, for algebraic specifications, we show that is is possible to automatically select test data sets from a structured specification. We have done a system developed in PROLOG, which allows to select a test data set from an algebraic specification and some hints about the chosen testing strategy. Our system is not described in this paper (it is described in

[Mar90] and [BGM90]), but we define the strategies used by the system as examples of the theory.

The paper is organized as follows:

- Obviously, you are reading the introduction (Section 1)...
- Section 2 contains an intuitive approach of our formalism
- Section 3 gives some preliminary notations and definitions
- Section 4 defines the fundamental notion of *testing contexts*
- Section 5 explains how testing contexts can be refined in order to produce "practicable" testing contexts
- Section 6 specializes to the case of *algebraic specifications*
- Section 7 shows our basic examples of testing strategies
- Section 8 explains how the problem of deciding success/failure of a test set is solved when the specification and the program are designed for testability
- Section 9 gives some hints on how to test less testable programs
- Section 10 recapitulates the main ideas of our theory.

2 Intuitive approach

We introduce the important idea (already sketched in [BCFG86]) that a *test data set* (i.e. a set of elementary tests) cannot be considered (or evaluated, or accepted, etc.) independently of:

- some *hypotheses* on the program which express the gap between the success of the test and the correctness of the program
- the existence of an *oracle*, i.e. a means of deciding, for any submitted test data, if the program behaves correctly with respect to its specification.

Thus, we define a *testing context* as a triple (H, T, O) where T is the test data set, H is a set of hypotheses and O is an oracle.

Then we state what is a *practicable* testing context, i.e. a context such that, assuming H, O is able to decide the success or failure of the test data set T and T is successful via O if and only if the program is correct. We then define a "canonical" testing context which, unfortunately, is rarely practicable and we provide some way of *refining* it until we get a practicable testing context.

Intuitively, a program P gives a way of computing some operations and a specification SP states some properties which should be satisfied by these operations. For example, assuming that P is supposed to implement three operations named *sin*, *cos* and *tg*, an axiom of the specification SP could be:

$$\textbf{If } x \equiv \frac{\pi}{2}\ [\pi] \textbf{ Then } cos(x) = 0 \textbf{ Else } tg(x) = \frac{sin(x)}{cos(x)}$$

Running one test of this axiom consists of replacing the variable x by some constant α, computing by P the compositions of operations which occur in the axiom (i.e. $cos(\alpha)$ and $tg(\alpha)$, $sin(\alpha)$ if needed) and deciding, via the oracle, if the axiom is validated by P. It means that the oracle has to decide if ($\alpha \equiv \frac{\pi}{2}\ [\pi]$), it has to use the application rules of "**If..Then..Else**" (as defined by the semantics of the specification language of SP) and to decide if, for instance, $cos(\alpha)$ is equal to 0 (where "*is equal to*" is also defined according to the semantics of the specification language).

When a *test data set* T is defined as a set of ground instances of such axioms, our view of program testing is just a generalization of the classical way of running tests: the program is executed for a given input, and the result is accepted or rejected, according to the axioms which play the role of input-output relations required for the program. Consequently, the correctness of these decisions is of first importance: if not ensured, it becomes possible to reject correct programs for instance. An oracle is some decision process which is able to decide, for each elementary test τ in T, if τ is successful or not when submitted to the program P. Thus, this decision is just a predicate, which will be denoted by O. Providing such an oracle is not trivial at all ([Wey80], [Wey82]) and may be impossible for some test data sets. Thus the existence of an oracle must be taken into account when selecting a test data set.

As usual, when a test data set is successful, the program correctness is not ensured, even if the oracle problem is solved: the set of all possible instances of the variables is generally infinite, or at least too large to be exhaustively experienced. However, when a test fails we know that the program is not correct: in some respects, it is consequently a success... Nevertheless, when the test is successful, we get a partial confidence in the program. We prefer to express this "partial confidence" as: "under what hypotheses does the success of the test data set imply the program correctness ?" These hypotheses are usually left implicit. We believe that it is of first importance to make them explicit. Moreover we consider that they are the good starting point for the selection of test data sets: it seems sound to state first the hypotheses and then to select a test data set which ensures correctness, assuming these hypotheses.

Thus, given a formal specification SP and a program P, the test data selection problem implies to state some hypotheses H and to select some test data set T such that, informally:

$$H + success(T) \Longleftrightarrow correctness(P, SP)$$

This equivalence can be shown as three implications:

- The *conservativity* means that the chosen hypotheses are satisfied by every correct program:

$$correctness \Longrightarrow H$$

- The *unbias* property means that a correct program cannot be rejected

$$correctness \Longrightarrow success$$

- The *validity* property means that under the hypotheses H, if the test is successful then the program is correct:

$$H + success \Longrightarrow correctness$$

Equivalently, it also means that under the hypotheses H, any incorrect program is discarded (similar to the completeness criteria of Goodenough and Gehrart [GG75]):

$$H + incorrectness \Longrightarrow failure$$

Another interesting view of the same implication is the following

$$incorrectness + success \Longrightarrow \neg H$$

which recalls that hypotheses are only... hypotheses. Intuitively, there is a negative correlation between the strength of the hypotheses and the size of the selected test data set. If the selected test data set does not reveal any error, the hypotheses may be too strong.

Notice that even when the hypothesis H is conservative, if the program P does not validate H, then the implication $H + success \Longrightarrow correctness$ is always true (!)[1]. However, one should not be confused. In this case, the success of the test does not imply correctness because $(A \text{ and } B) \Longrightarrow C$ is not equivalent to $B \Longrightarrow C$, specially when A is false... the implication is simply meaningless: the hypotheses are too strong.

Summing up this section, we have shown that it is not sufficient to simply select a test data set T. One should provide a related set of hypotheses H and a well defined oracle O, with the property:

$$H + \text{ success of } T \text{ via } O \Longleftrightarrow correctness$$

These three components are formally defined in Section 4. Let us first state several convenient notations.

3 Notations

We consider a rather flexible definition of "formal specification." It embeds various approaches of formal specifications such as algebraic specifications, temporal logic and other kinds of logic, etc. The readers familiar with Goguen and Burstall institutions will recognize a simplified version of them.

A formal specification method is given by a *syntax* and a *semantics*.

- The syntax is defined by a class of *signatures* and a set of *sentences* is associated to each signature.

  In practice, a signature Σ is a set of operation names. Given a signature Σ, the associated set of Σ-sentences, denoted by Φ_Σ, contains all the well-formed formulas built on: the operations in Σ, some variables, some atomic predicates and some logical connectives.

[1] I must thank one of the referees for this remark.

- The semantics is defined as follows: for each signature Σ there is an associated class of Σ-models, denoted by Mod_Σ, and there is a "validation predicate" on $Mod_\Sigma \times \Phi_\Sigma$, also called "the *satisfaction relation*", denoted by "$\models$". For each Σ-model $M \in Mod_\Sigma$ and for each Σ-sentence $\varphi \in \Phi_\Sigma$, "$M \models \varphi$" should be read as "M validates φ."

In this framework, a *formal specification* is a couple $SP = (\Sigma, Ax)$ such that Ax is a finite subset of Φ_Σ. The models of Mod_Σ which validate all the sentences of Ax are the models *validating* (or *satisfying*) SP. We denote by $Mod(SP)$ this class of models:

$$Mod(SP) = \{\ M \in Mod_\Sigma \mid (\forall \varphi \in Ax)(\ M \models \varphi\ )\ \}$$

Of course the notions of signature, sentence, model and satisfaction relation depend on the kind of formal specification one need to consider. It is possible to verify the adequacy or inadequacy of a program P with respect to a specification SP only if the semantics of P and SP are expressible in some common framework. Thus the notion of model must be carefully defined.

When the signature Σ is just a set of operation names, a Σ-model M is usually a set of values and for each operation name of Σ, there is an operation of the relevant arity in M. Then, as a program gives a way of computing operations, one can consider that the behaviour of a program P defines a Σ_P-model, where Σ_P is the set of the names of all the operation exported by P.

It is important to note here that the variables, atomic predicates and logical connectives allowing to built the formulas of SP do not belong to the signature Σ. For instance SP may contain existential quantifiers while P does not (and this is a usual case).

The model M_P associated with P is not well known *a priori*. It is the reason why validation techniques such as testing or proving must be used in order to check whether $M_P \in Mod(SP)$ (as "P is correct" is equivalent to "$M_P \in Mod(SP)$").

4 Testing contexts

Definition 1 : Let P be the program under test. Let $SP = (\Sigma, Ax)$ be the specification that P is supposed to implement. A *testing context* is a triple (H, T, O) where:

- H is a set of hypotheses about the model M_P associated with P. This means that H describes a class of models $Mod(H)$. $Mod(H)$ is a subclass of Mod_{Σ_P} where Σ_P is the signature associated with P ($Mod(H)$ is called the class of models "satisfying H")
- T is a subset of Φ_Σ. It is called a "test data set" and each element τ of T is called an "elementary test"
- the oracle O is a partial predicate on Φ_Σ. For each sentence (think "each elementary test") φ in Φ_Σ , either $O(\varphi)$ is undefined either it decides if φ is successful when submitted to P.

This definition is very general and call for some comments:

The signature Σ_P associated with P contains, in practice, the names (and arities) of all the operations exported by P (as explained in Section 3 above). A priori we do not assume any adequacy between Σ_P and Σ (but we shall do it soon). Indeed, $Mod(H)$ will characterize a subclass of programs such that "incorrectness" implies "failure of the test."

It may seem surprising that test data sets can contain sentences with variables. Of course, our aim is to select test sets which are: executable (i.e. containing *ground* sentences), finite and mainly instances of the axioms of SP. However, the definition above is useful because it allows to build testing contexts by refinements.

The oracle O can be shown as a procedure using the program P: one should write O_P but for clarity of the notations P and SP are implicit parameters of the testing contexts.

Of course the goal of testing context refinements is to get so called practicable testing contexts:

Definition 2 : Let (H, T, O) be a testing context.

1. (H, T, O) *has an oracle* means that:
   - T is finite (think "of reasonable size")
   - if M_P satisfies H then T is included in $D(O)$, where $D(O)$ is the definition domain of O (i.e. O is defined for each element of T)
   - if M_P satisfies H then O is decidable for each element τ of T.
2. (H, T, O) is *practicable* means that:
   - (H, T, O) has an oracle
   - if M_P satisfies H then: $O(T) \Longleftrightarrow M_P \models Ax$
     where $O(T)$ denotes "$O(\tau)$ for all τ in T" which means that the test data set is successful via the oracle. Notice that $O(T)$ is defined and decidable when (H, T, O) has an oracle.

Let us describe some special examples of testing contexts.

The proof examples: Let us assume that the syntaxes and semantics of P and SP allow theorem proving (e.g. P is a rewrite rule system and SP is an equational specification). One can consider the testing context where T is equal to Ax (the axioms of SP) and O is some theorem prover (e.g. by structural induction). One should notice that, even for this purely proof oriented example, the corresponding set of hypotheses H is not empty. In the example where P is a set of rewrite rules and O is based on structural induction methods, H must contain at least two hypotheses: the signature of P and SP are equal (i.e. $Mod(H) \subseteq Mod_\Sigma$) and every term submitted to P must be build on the operations of the signature (i.e. $Mod(H)$ only contains finitely generated models). Such hypotheses must not be neglected (more than 50% of the *proved* factorials admit negative arguments and loop on them !). The main advantage of our approach is that the hypotheses are *explicit*.

Such "proof oriented" testing contexts are practicable if and only if every axiom of SP is decidable via the theorem prover O.

The "lazy example:" One can also consider an example where the hypothesis is "P is correct" (i.e. $Mod(H) = Mod(SP)$). Then T is empty and O is the undefined predicate. It reflects at least a full confidence in the program design. For instance, it can be used when the program has been automatically build from the specification, the hypothesis simply means that the program construction system is supposed correct. Most of the time, it seems sound to (try to) prove such hypotheses...

Anyway, these testing contexts are always practicable.

The exhaustive test set: Under the same set of hypotheses H as for the proof oriented case, one can consider the test data set T containing all ground instances of Ax. Assuming that there exists a decidable oracle for all ground instances, the resulting testing context is practicable if and only if T is finite (which is not the general case). Such an exhaustive test data set may work for enumerated types for instance.

Since the exhaustive test data set is generally infinite; since the empty test data set gives rise to an hypothesis which is too strong; since proving correctness is often too costly, it is clear that a good testing context is something in the middle of these three extremist views of testing. Such good and practicable testing contexts can be obtained via "testing contexts refinements." A natural starting point of these refinements is what we call the canonical testing context. It is build from the informations that we directly get from P and SP:

Definition 3 : The *canonical testing context* is defined by:

- the hypothesis "M_P is a Σ-model" which means $\Sigma_P = \Sigma$, i.e. $Mod(H) = Mod_\Sigma$
- the test data set $T = Ax$
- the oracle $O = undef$ (the never defined predicate)

Let us give some comments. It is not difficult to check in practice that the signature of P is Σ: it is just the set of exported operations implemented by P. Considering the axioms of SP as the canonical test data set simply means that our goal is to check whether P is compatible with the specification. Of course, the refinement process allows to select, step by step, a finite set of executable elementary tests. Similarly, the oracle, which is unknown at the starting point, is refined until it is decidable on the (refined) test data set. The crucial property of the canonical testing context is that it is valid (as proved in the next section).

Indeed, one need to define the validity and unbias properties (already sketched in Section 2): they provide sufficient conditions for practicability.

Definition 4 : Let (H, T, O) be a testing context.

1. The test data set T is *valid* means that:
   If P satisfies H (i.e. $M_P \in Mod(H)$) then

$$M_P \models T \implies M_P \models Ax$$

2. The oracle O is *valid* means that:
   If $M_P \in Mod(H)$ then

$$(\forall \varphi \in D(O))(\ O(\varphi) \Longrightarrow M_P \models \varphi\ )$$

3. The test data set T is *unbiased* means that:
   If $M_P \in Mod(H)$ then

$$M_P \models Ax \ \Longrightarrow\ M_P \models T$$

4. The oracle O is *unbiased* means that:
   If $M_P \in Mod(H)$ then

$$(\forall \varphi \in D(O))(\ M_P \models \varphi \Longrightarrow O(\varphi)\ )$$

The sufficient condition for practicability is given by the following theorem:

Theorem 5 : Let (H, T, O) be a testing context. If (H, T, O) has an oracle and T and O are both valid and unbiased, then (H, T, O) is practicable.

The proof of this theorem is trivial. However, this theorem is fundamental since it justifies the testing refinement process: the canonical testing context defined above is valid and unbiased; in the following section we give a refinement criterion which preserves validity; and we give another criterion, for algebraic specifications, which ensures unbias. Consequently, practicability is ensured providing that we stop the refinement process when the triple (H, T, O) has an oracle (we shall explain how to get this result in a finite number of refinement steps).

5 The refinement preorder

Definition 6 : Let $TC_1 = (H_1, T_1, O_1)$ and $TC_2 = (H_2, T_2, O_2)$ be two testing contexts. TC_2 *refines* TC_1, denoted by $TC_1 \leq TC_2$, means that:

- $Mod(H_2) \subseteq Mod(H_1)$, (i.e. $H_2 \Longrightarrow H_1$)
- $(\forall M_P \in Mod(H_2))$ ($M_P \models T_2 \Longrightarrow M_P \models T_1$)
- if $M_P \in Mod(H_2)$ then $D(O_1) \subseteq D(O_2)$ and

$$(\forall \varphi \in D(O_1))\ (\ O_2(\varphi) \Longrightarrow O_1(\varphi)\ )$$

The first condition means that it is possible to encrease the hypotheses about the program under test via a refinement step. As already outlined in Section 2, encreasing the hypotheses allows to decrease the size of the selected test data set without loosing validity. This idea is reflected by the second condition. It exactly means that *under the new hypotheses* (H_2, which can be bigger than H_1), the new test data set T_2 must reveal an error if T_1 reveals an error. Examples where encreasing H allows to decrease T are given in Section 7. The third condition means that the oracle predicate can be build, step by

step, along the refinement preorder and that every error revealed by O_1 must be revealed by O_2.

Notice that the refinement defined above is clearly a preorder (reflexive and transitive); it is not antisymmetric.

The following theorem is trivial, but it is important because it ensures the validity of the test data set, providing that the refinement starts from the canonical testing context defined in Section 4 above:

Theorem 7 : Let (H, T, O) be a testing context. If (H, T, O) refines the canonical testing context then T is valid.

By the way, the refinement preorder allows trivially to express all the interesting properties defined in Section 4:

Proposition 8 : Let (H, T, O) be a testing context. Let $Satisf_{D(O)}$ be the partial predicate on Φ_Σ defined by: the definition domain of $Satisf_{D(O)}$ is $D(O)$ and for each φ in $D(O)$, $Satisf_{D(O)}(\varphi)$ is equivalent to $M_P \models \varphi$. It simply means that $Satisf_{D(O)}$ coincides with the satisfaction relation "$\models$" in the definition domain of O.

1. The oracle O is valid if and only if $(H, T, Satisf_{D(O)}) \leq (H, T, O)$
2. The test data set T is unbiased if and only if $(H, T, O) \leq (H, Ax, O)$ (where Ax is the set of axioms of SP)
3. The oracle O is unbiased if and only if $(H, T, O) \leq (H, T, Satisf_{D(O)})$.
4. The hypotheses H are conservative iff $(H, T, O) \leq (Correct, T, O)$, where $Correct$ is the hypothesis defined by $Mod(Correct) = Mod_\Sigma(SP)$.

From *(1.)* and *(3.)* one can deduce that the oracle problem is to exhibit a decidable subdomain of the satisfaction relation which covers the selected test data set. Moreover, if the formal specification method under consideration has some Birkhoff's property, *(2.)* means that one must select elementary tests which are theorems of the axioms of SP.

In the rest of this paper, we show how to fulfill these criteria. For that purpose, we specialize to *algebraic specifications*.

6 Case of algebraic specifications

In this section, we first recall the main definitions of *algebraic abstract data types* as an instance of our general definition of formal specifications. Then we show that valid and unbiased test data sets can be selected from the so called "exhaustive test data set."

In the framework of algebraic specifications, a signature is: a finite set S of *sorts* (i.e. type-names) and a finite set of *operation-names* with arity is S. The corresponding class of models Mod_Σ is defined as follows: a Σ-model is a heterogeneous set M partitioned as $M = \{M_s\}_{s \in S}$, and with, for each operation-name "$op : \; s_1 \times \cdots \times s_n \to s$" in Σ, a total function $op^M : \; M_{s_1} \times \cdots \times M_{s_n} \to M_s$.

The sentences of Φ_Σ are the positive conditional equations of the form:

$$(v_1 = w_1 \ \wedge \cdots \wedge \ v_k = w_k) \implies v = w$$

where v_i, w_i, v and w are Σ-terms with variables ($k \geq 0$). A model M satisfies ($\models$) such a sentence if and only if for each substitution σ with range in M, if $\sigma(v_i) = \sigma(w_i)$ for all i then $\sigma(v) = \sigma(w)$ (as in [ADJ76]).

Moreover specifications are structured: a *presentation* (or "specification module") $\Delta SP = (\Delta\Sigma, \Delta Ax)$ uses some *imported* specifications $SP_i = (\Sigma_i, Ax_i)$ such that $SP = \Delta SP +$ (union of SP_i) is a (bigger) specification. The signature Σ of SP is the disjoint union of $\Delta\Sigma$ and the union of the Σ_i; the set of axioms Ax of SP is the union of ΔAx and the Ax_i . For example, a *List* presentation (with a sorted insertion) over the imported specification of natural numbers can be expressed as in Figure 6 (where n and m are

---

1— A list presentation

$\Delta NATLIST$ uses NAT /* Here ΔNAT uses $BOOLEAN$ */
$\Delta S =$ $\{ NatList \}$
$\Delta\Sigma =$ $empty : \rightarrow NatList$
$cons : Nat \times NatList \rightarrow NatList$
$ins : Nat \times NatList \rightarrow NatList$
$\Delta Ax =$ $ins(n, empty) = cons(n, empty)$
$n \leq m = true \Longrightarrow ins(n, cons(m, L)) = cons(n, cons(m, L))$
$n \leq m = false \Longrightarrow ins(n, cons(m, L)) = cons(m, ins(n, L))$

---

variables of sort Nat; L of sort $NatList$).

Of course, all the results stated in the previous sections remain for the particular case of structured algebraic specifications. In particular, given a testing context (H, T, O), an elementary test of T can be any positive conditional equation. The following definitions and results prove that it is possible to select ground instances of the axioms without loosing validity. Moreover, in that case, the unbias property is ensured.

Definition 9 :

1. A Σ-model M is *finitely generated* if and only if every value of M is denotable by a ground Σ-term. We denote by $Adequate\Sigma$ the hypothesis such that $Mod(Adequate\Sigma)$ is the class of finitely generated Σ-models.

   This hypothesis means that the signature exported by P is exactly the signature of SP and that P does not admit inputs which are not denotable via Σ (we have already mentioned this hypothesis in the "proof example" given in Section 4).
2. Given a specification SP, the *exhaustive test data set*, $Exhaust_{SP}$, is the set of all ground instances of all the axioms of SP.

   $$Exhaust_{SP} = \{ \sigma(\varphi) \mid \varphi \in Ax \ , \ range(\sigma) = W_\Sigma \}$$

   where W_Σ is the set of ground terms on Σ.

Under the Σ-adequacy hypothesis $Adequate\Sigma$, every testing context which refines, and is included in the exhaustive test data set, produces valid and unbiased test data sets. More precisely:

Proposition 10 : Let (H, T, O) be a testing context.

1. If $T \subseteq Exhaust_{SP}$ then T is unbiased.
2. If (H, T, O) refines $(Adequate\Sigma, Exhaust_{SP}, undef)$ then T is valid.

Proposition *(1.)* is trivial. Proposition *(2.)* results from the fact that the context $(Adequate\Sigma, Exhaust_{SP}, undef)$ refines the canonical testing context defined in Section 4 and from the validity theorem of Section 5.

In particular the exhaustive test data set itself is unbiased and valid. Unfortunately it is generally infinite. The previous propositions means that the test data set refinement process can be reduced to "add hypotheses to $Adequate\Sigma$ in order to select a subset of reasonable size from $Exhaust_{SP}$."

In the next section, we show via an example what kind of hypotheses can be used. Our system handles the corresponding refinements automatically.

7 Regularity and uniformity

Let us assume that we want to test the axiom:

$$n \leq m = false \Longrightarrow ins(n, cons(m, L)) = cons(m, ins(n, L))$$

One has to select instances of the list variable L and instances of the natural variables m and n. A first idea, which is often used when testing, is to bound the size of the lists actually tested. It means to bound the size of the terms substituting L. This can be obtained via the general schema of *regularity hypothesis*.

Definition 11 : Let $\varphi(L)$ be a sentence involving a variable L of a sort $s \in \Delta S$. Let $|t|_s$ be a complexity measure on the terms t of sort s (for instance the number of operations in $\Delta\Sigma$ of sort s occurring in t). Let k be a positive integer. The *regularity hypothesis* of level k (with respect to φ and L), $Regul_{\varphi(L),k}$, is the hypothesis which retains the Σ-models M_P such that:

$$(\forall t \in W_\Sigma)(\ |t|_s \leq k \Longrightarrow M_P \models \varphi(t)\ ) \Longrightarrow (\forall t \in W_\Sigma)(\ M_P \models \varphi(t)\ )$$

where W_Σ is the set of all ground Σ-terms.

Instead of a complexity measure, any function such that the sets $\{t \in W_\Sigma \mid\ |t|_s \leq k\}$ are finite is acceptable. At first glance, it may seem unreasonable to use hypotheses which infer a result for every list by checking only small lists. Anyway, these hypotheses reflect what is done when testing. The main advantage of our approach is to make them *explicit*. Moreover one should never forget that the goal of testing is not to prove correctness, rather the goal of testing is to find good potential counter-examples of the correctness, as discussed in Section 2.

For the list example, a regularity level 3 allows to select the following instances of L:

$L = empty$
$L = cons(p, empty)$
$L = ins(p, empty)$
$L = cons(p, cons(q, empty))$
$L = ins(p, cons(q, empty))$
$L = cons(p, ins(q, empty))$
$L = ins(p, ins(q, empty))$

The corresponding refinement step of the testing context is obtained by adding the regularity hypothesis to H, and by replacing in T the axiom under test by the seven axioms related to those seven instances of L. Consequently, only variables of sort natural number remain: m, n, p and q.

Notice that the four instances of L involving the operation *ins* seems to be less relevant than those involving only *cons* and *empty* because *cons* and *empty* generate all the list values. Such a subset $\Delta\Omega$ of $\Delta\Sigma$ is called a set of generators. It is not difficult to modify the regularity hypothesis in order to get a so called Ω-regularity hypothesis which only retains the three instances of L build on *cons* and *empty* (as defined in [BGM90]).

At this stage in the example, or after a finite number of regularity hypotheses in the general case, the test selection problem is reduced to the replacement of the variables of imported sort (natural number) by ground terms. The interesting cases correspond to the 4!=24 relative orders of m, n, p and q. These 24 arrangements provide 24 subdomains of $\mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N} = \mathbb{N}^4$. A good testing strategy is to select one value in each subdomain. This can be obtained using *uniformity hypotheses* on these 24 subdomains.

Definition 12 : Let $\varphi(V)$ be a formula involving a variable, or a vector of variables, V of imported sort(s). Let SD be a subdomain (i.e. a subset) of $W_{\Sigma,s_1} \times \cdots \times W_{\Sigma,s_k}$, where $s_1 \cdots s_k$ are the sorts of the variables of V. The *uniformity hypothesis* on the subdomain SD (with respect to φ and V), $Unif_{\varphi(V),SD}$, is the hypothesis which retains the Σ-models M_P such that:

$$(\forall v_0 \in SD)(\ M_P \models \varphi(v_0) \Longrightarrow [\forall v \in DS][M_P \models \varphi(v)]\ )$$

Such an hypothesis means that if a sentence is true for some value v_0 then it is always true in DS... This is a strong hypothesis, and may seem unreasonable again. However, the same arguments hold: the hypotheses reflect what is done when testing; we have just made them *explicit*; and the goal is not to prove correctness, it is to find potential counter-examples of correctness.

A more interesting question is: "how to find the relevant subdomains ?". Our system is able to find them automatically. The key is: unfolding methods (based on equational logic programming with constraints). These methods are described in [Mar90] or [BGM90].

Notice that the modularity of the specification is crucial here: the regularity hypotheses are made for the sorts specified by ΔSP while the uniformity hypotheses are made for the imported sorts.

Very roughly, our system starts from a modular specification and translates the axioms into an equivalent set of Horn clauses ("equivalent" is defined in [Mar90][BGM90]). Then, for each axiom of ΔAx, the system selects a test data set as follows: for each sort of ΔS involved in the axiom, the system adds clauses reflecting the regularity hypotheses (the user gives the regularity level k); the axiom under test is transformed, via the regularity clauses, into a set of axioms where only variables of imported sorts remain; the unfolding methods are then applied to each transformed axiom, leading to a set of predicates (via the "wait" mechanisms) which define the uniformity subdomains; one instance of each subdomain is then selected (by resolution with a random choice of the clauses to apply). Of course, many extensions of equational logic programming have been used, completeness being one of the main difficulties. The system is fully described in [Mar90]; a simpler description can be found in [BGM90].

8 The oracle

In this section, we explain how the oracle problem can be solved using observability issues from the specification level.

Hypotheses such as those exposed in the previous section provide test selection strategies which allow to select finite test data sets included in $Exhaust_{SP}$. Thus, an elementary test is of the form:

$$(t_1 = u_1 \quad \wedge \cdots \wedge \quad t_k = u_k) \Longrightarrow t = u$$

where t_i, u_i, t and u are ground Σ-terms. The oracle problem is then reduced to decide success/failure of equalities between ground terms, because the truth tables of $\wedge$ and $\Longrightarrow$ can then be used to decide success/failure of the whole conditional elementary test.

It is well known that deciding abstract equalities is not trivial at all. For example, let P implement stacks by means of arrays: *push* records the element at range *height* into the array and increments *height*; *pop* simply decreases *height*. Suppose the oracle has to decide if $pop(push(3, empty))$ and *empty* give the same stack after their executions via P. There is an observability problem which, indeed, is a concrete equality problem: these stacks give two distinct array representations (because 3 has been recorded into the array for the first stack) but they are abstractly equal (as the common height is 0, and there is no observable access, via *top*, which allows to distinguish them). Thus, the predicate $O(pop(push(3, empty)) = empty)$ must be decided via some carefully elaborated method.

This problem relies on observability issues. Given a structured specification SP (e.g. *Stack*), a subset S_{Obs} of *observable sorts* is distinguished among the imported sorts (e.g. the natural numbers and the elements of the stacks). These observable sorts are specified by imported *observable specifications* SP_{Obs1} , SP_{Obs2}, etc. Intuitively, they correspond to the observable primitive types of the used programming language. In particular, for every program P, there are built-in equality predicates which are correct and defined on observable sorts.

There are mainly three approaches to handle the oracle problem, which are discussed in the rest of this paper.

A first idea is to modify the program under test in order to add new procedures

extracting the concrete representations and computing the abstract equalities for non observable sorts. However, obviously, some of the advantages of the black-box approach are lost and the added procedures may alter the program behaviour. Then, the actually tested behaviour could not be represented by M_P any more, which results in a biased application of our formalism.

Indeed the proper approach is to take into account, from the specification stage, all these equality predicates for non observable sorts. This correspond to the well known slogan "plane *testable* programs at every stage of the design process." In this case, the oracle equality predicates are operations of Σ and our formalism can be directly applied.

However, since the oracle equalities are crucial for the quality of the performed tests, any hypothesis on them is very strong. Consequently, we recommand to put the sentences related to those equality predicates in the "proof part" of the testing context (Section 4). Then, the hypotheses (such as regularity and uniformity) are made only on the other sentences of the specification. This is "the good solution" for the oracle. However, when the program has not been designed to be easily testable, some partial results can be obtained via the second and third approaches which are discussed in the next section.

9 Partial oracles

The second approach is to select only observable elementary tests. This can be done by replacing each non observable test $[t = u]$ by a set of terms of the form $[C(t) = C(u)]$ obtained by surrounding t and u with some well chosen *observable contexts* C. Here, the word "context" must be understood as "term with exactly one variable."

For example, an equality between two stacks $[t = u]$ seems to be equivalent to the following observable equalities: $[height(t) = height(u)]$, $[top(t) = top(u)]$, $[top(pop(t)) = top(pop(u))]\ldots[top(pop^{height(t)-1}(t)) = top(pop^{height(t)-1}(u))]$. Here the chosen observable contexts C are $height(x)$ and the $top(pop^i(x))$ such that $i < height(t)$. Unfortunately, it is well known that identifying such a minimal set of contexts is undecidable for the general case of algebraic specifications [Sch86]. Besides, when testing "big" stacks, this leads to an impracticable number of observable tests.

But the situation is even worst: when we think that $height(x)$ and $top(pop^i(x))$ are sufficient, we implicitly assume that we manipulate (more or less) stacks. But this fact is just what we check when testing !

Counter-example: We sketch a program which does not satisfy this implicit hypothesis. The "bug" is that *top* returns the height when applied to a term t of the particular form $t = push(e, pop(\cdots))$; for every other term t, $top(t)$ returns the correct value. A "stack" is implemented by a record `<array,height,foo>` where `<array,height>` is the usual correct implementation and `foo` records the number of `push` performed since the last `pop` (and the empty stack is initialized with `foo=2`). We ignore the exceptional cases of `pop` and `top` when the stack is empty, it is not our purpose here...

```
proc emptystack();
```

```
        stack.height := 0 ; stack.foo := 2 ;;

    proc push(e:element);
        stack.array[stack.height] := e ;
        stack.height := stack.height+1 ; stack.foo := stack.foo+1
       ;;

    proc pop();
        stack.height := stack.height - 1 ; stack.foo := 0 ;;

    proc top();
        if (stack.foo = 1) then return stack.height    /* the bug */
        else return stack.array[stack.height] ;;
```

The terms $t = push(1, emptystack)$ and $u = pop(push(2, push(1, emptystack)))$ are distinguishable because we get `top(push(0,t)) = 0` (as `foo=4`) and we get `top(push(0,u)) = 2` (as `foo=1`). Nevertheless, the contexts $height(x)$ and $top(pop^i(x))$ are unable to distinguish t from u (as `foo` is never equal to 1 for those contexts), leading to an oracle which never detects that $t \neq u$.

So, we get the depressing result that identifying the minimal set of observable contexts for the equality decision *is decidable* ! We must consider the set of *all* observable contexts... which is infinite, consequently impracticable.

We can follow a similar method as for test selection: this infinite set of observable contexts can be reduced to a finite one by adding oracle hypotheses in the component H of the testing context. For example, the finite set of contexts mentioned for stacks is related to the following hypothesis on M_P:

> "for all stacks t and u, if M_P satisfies $(height(t) = height(u))$ and $(top(pop^i(t)) = top(pop^i(u)))$ for $i < height(t)$ then M_P satisfies $(t = u)$."

It will be useful later to remark here that the number of observable contexts of the form $top(pop^i(x))$ is decided from the specification before the test is performed. Thus $i < height(t)$ should be understood with respect to the specification, $height(t)$ is not computed by the program here. Of course, the counter-example above does not meet this hypothesis and it will not be discarded by the selected observable test data set. However, one time more: the main advantage of our approach is to make the testing hypotheses *explicit*.

This idea of adding oracle hypotheses works when deciding equalities which appear *in the conclusion* of the tested conditional sentences. Unfortunately, if the equality appears in the precondition, for instance:

$$t = u \Longrightarrow concl$$

then we get a biased oracle ! This results from: $t = u$ may be successful according to the oracle hypotheses, but not satisfied in M_P. In that case, one would require for *concl* to be satisfied, in spite of the fact that *concl* is not required according to the

formal satisfaction relation. Of course a first case where the solution is trivial is when the specification only contains sentences with observable preconditions (it is not really restrictive in practice). Another case which allows to solve this problem is when the specification is "complete" with respect to the imported observable specifications: if the oracle hypotheses are conservative then the oracle remain unbiased. All these techniques are more precisely studied in [Ber89].

The third approach, somehow intermediate between the two first approaches, is an oracle which "drives" the program under test. The oracle is then realized as a new module on the top of the program under test. The new operations of this module are equalities for the non observable sorts where an equality predicate does not already exist in Σ_P. Notice that these oracle equalities are not "added into the code of P," rather they are computed on the top of P, and they can only exploit the observable results of P. For our stack example, we may add an equality predicate for the sort stack on the top of P as follows:

```
function eq(t,u:stack): boolean;
    If (height(t) ≠ height(u)) then return false
    Else
        If (height(t)=0) then return true
        else return ( (top(t)=top(u) and eq(pop(t),pop(u)) ) ;;
```

By convention, the terms which are computed by P have been written in `typewriter type style`.

Of course, this oracle is not able to discard the counter-example given above. This means that this oracle is valid (as defined in Section 4) under an additional oracle hypothesis. This hypothesis is similar to the hypothesis expressed for the second approach, except that the number of observable contexts of the form $top(pop^i(x))$ is dynamically computed by P: it depends on `height(t)` as computed by P; it does not depend on $height(t)$ as specified by SP.

The difficult point here is that one must *prove* that the computed oracle equalities are sound with respect to the oracle hypotheses. If possible, we recommand to define them via rewrite rules because rewrite rules facilitate proofs in the framework of algebraic specifications. This approach is more detailed in [Ber89].

Anyway, even if the approaches sketched in this section have some theoretical interest, practical experiments showed that they are not realistic: observable contexts (resp. oracle equalities) are often complex and there is a big number of them (for specifications of real size). This leads to an amount of potential errors which is almost comparable to the amount of faults in the program under test. It is far more suitable to plane testable programs from the specification stage, as recommended in Section 8.

10 Recapitulation

This paper gives a formal study of testing when a formal specification of the program under test is available. The main advantage is to make explicit, in a well defined framework,

the limitations and the abilities of testing.

We have shown that a *test data set* cannot be considered, or evaluated, independently of: some *hypotheses* which express the gap between the success of the test data set and the correctness of the program; and the existence of an *oracle* which is a means of deciding success/failure of the test set. Thus, we have defined the fundamental notion of *testing context* which reflects theses three components.

Our definition of testing context is powerful enough to reflect a software validation approach where some of the properties required by the specification are proved, while the other ones are tested. This allows to mix proof methods and testing methods in a unified framework.

We have described, and formally justified, a refinement method which allow to get so called *practicable* testing contexts. Practicable testing contexts have finite test data sets and decidable oracles, and they have the following crucial property:

$$\text{Hypotheses} + \text{success of test via oracle} \Longleftrightarrow \text{correctness}$$

When *algebraic specifications* are involved, we have established several more precise results which lead to a concrete way of building practicable testing contexts by refinements.

Several other works based on the theory described in this paper have been done, or are investigated:

We have done a system, based on equational logic programming with constraints, which automatically selects a practicable test data set from the specification and some hints about the corresponding hypotheses. Its existence proves that our formalism is usable.

We also extend our formalism to the case of algebraic specification with exception handling. It shall allow, in particular, to handle bounded data structures where performing tests "near the bounds" is crucial.

Acknowledgements: Obviously, the work reported here is only the "basic theoretical part" of common researches about testing via formal specifications, and it is a pleasure to acknowledge *Marie Claude Gaudel*, *Bruno Marre* and *Pascale Le Gall* for numerous interactions and discussions which have allowed to build this formalism. This work has been partially supported by the Meteor Esprit Project and the PRC "Programmation et Outils pour l'Intelligence Artificielle."

References

[ADJ76] J. Goguen, J. Thatcher, E. Wagner : "*An initial algebra approach to the specification, correctness, and implementation of abstract data types*", Current Trends in Programming Methodology, Vol.4, Yeh Ed. Prentice Hall, 1978.

[BCFG85] L. Bougé, N. Choquet, L. Fribourg, M. C. Gaudel : "*Application of PROLOG to test sets generation from algebraic specifications*", Proc. International Joint Conference on Theory and Practice of Software Development (TAPSOFT), Berlin (R.F.A), Springer-Verlag LNCS 186, pp.246-260, March 1985.

[BCFG86] L. Bougé, N. Choquet, L. Fribourg, M. C. Gaudel : "*Test sets generation from algebraic specifications using logic programming*", Journal of Systems and Software Vol 6, No.4, pp.343-360, November 1986.

[BGM90] G. Bernot, M. C. Gaudel, B. Marre : "*Software testing based on formal specifications: a theory and a tool*", To appear in Software Engineering Journal, U.K., 1991. (also: Internal Report LRI No.581, Orsay, France, June 1990.)

[Ber89] G. Bernot : "*A formalism for test with oracle based on algebraic specifications*", LIENS Report 89-4, LIENS/DMI, Ecole Normale Supe'rieure, Paris, France, May 1989.

[Bou85] L. Bougé : "*A proposition for a theory of testing: an abstract approach to the testing process*", Theoretical Computer Science 37, North-Holland, 1985.

[GCG85] C. P. Gerrard, D. Coleman, R. Gallimore : "*Formal Specification and Design Time Testing*", Software Science ltd, technical report, June 1985.

[GG75] J. B. Goodenough, S. L. Gerhart : "*Towards a theory of test data selection*", IEEE trans. soft. Eng. SE-1, 2, 1975. (Also: SIGPLAN Notices 10 (6), 1975.)

[GHM81] J. Gannon, P. McMullin, R. Hamlet : "*Data-Abstraction Implementation, Specification, and Testing*", ACM transactions on Programming Languages and Systems, Vol 3, no 3, July 1981, pp.211-223.

[GM88] M. C. Gaudel, B. Marre : "*Algebraic specifications and software testing: theory and application*", Internal Report LRI 407, Orsay, France, February 1988, and extended abstract in Proc. workshop on Software Testing, Banff, IEEE-ACM, July 1988.

[Mar90] B. Marre : "*Sélection automatique de jeux de tests a partir de spécifications algebriques, en utilisant la programmation logique*", Ph. D. Thesis, LRI, Université de Paris XI, Orsay, France, January 1990.

[Rig85] G. Rigal : "*Generating Acceptance Tests from SADT/SPECIF*", IGL technical report, August 1986.

[Sch86] O. Schoett : "*Data abstraction and the correctness of modular programming*", Ph. D. Thesis, Univ. of Edinburgh, 1986.

[Scu88] G. T. Scullard : "*Test Case Selection using VDM*", VDM'88, Dublin, 1988, LNCS no 328 pp 178-186.

[Wey80] E. J. Weyuker : "*The oracle assumption of program testing*", Proc. 13th Hawaii Intl. Conf. Syst. Sciences 1, pp.44-49, 1980.

[Wey82] E. J. Weyuker : "*On testing non testable programs*", The Computer Journal 25, 4, pp.465-470, 1982.

Chapitre 4 :
Software testing based on formal specifications: a theory and a tool

Gilles BERNOT
LIENS, CNRS URA 1327
Ecole Normale Supérieure
45 Rue d'Ulm
F-75230 PARIS Cédex 05
FRANCE
bitnet: bernot@frulm63
uucp: bernot@ens.ens.fr

Marie Claude GAUDEL
LRI, UA CNRS 410
Université PARIS-SUD
Bât. 490
F-91405 Orsay cedex
FRANCE
bitnet: mcg@frlri61
uucp: mcg@lri.lri.fr

Bruno MARRE
LRI, UA CNRS 410
Université PARIS-SUD
Bât. 490
F-91405 Orsay cedex
FRANCE
bitnet: marre@frlri61
uucp: marre@lri.lri.fr

(To appear in Software Engineering Journal, 1991)

Contents

1 Formal specifications and testing **99**

1.1 Testing a program against its formal specification 99

1.2 Testing contexts 101

1.3 The refinement preorder 103

2 Algebraic specifications and testing **104**

2.1 Algebraic specifications 104

2.2 The exhaustive test set 105

2.3 The oracle 108

2.4 Reducing test to equalities 112

3 Choice of regularity and uniformity hypotheses **115**

3.1 The problem 115

3.2 Decomposition of subdomains by unfolding defined operations 116

4 Basic tools for automatizing test data selection 119

4.1 Transforming axioms into Horn clauses 121

4.1.1 Transformation rules for the first step 121

4.1.2 **Correctness of the transformation** 122

4.1.3 Last step of the transformation 123

4.1.4 Examples 123

4.1.5 SLD resolution versus equational reasoning 124

4.2 Using a complete search strategy 124

4.3 The termination problem 125

4.4 Random choice of clauses 126

4.5 Decomposition into uniformity subdomains 127

4.6 Strategies for regularity and uniformity hypotheses 128

5 An example of test data set selection using our system 130

5.1 Overview of the system 130

5.2 Examples of test data set selections 131

5.2.1 First defining axiom of sorted 131

5.2.2 Second defining axiom of sorted 132

5.2.3 Third defining axiom of sorted 132

Abstract

This paper addresses the problem of constructing test data sets from formal specifications. Starting from a notion of an ideal exhaustive test data set which is derived from the notion of satisfaction of the formal specification, it is shown how to select by refinements a *practicable* test set, i.e. computable, not rejecting correct programs (*unbiased*), and accepting only correct programs (*valid*), assuming some hypotheses.
The hypotheses play an important role: they formalize common test practices and they express the gap between the success of the test and correctness ; the size of the test set depends on the strength of the hypotheses.
The paper shows an application of this theory in the case of algebraic specifications and presents the actual procedures used to mechanically produce such test sets, using Horn clause logic. These procedures are embedded in an interactive system which, given some general hypotheses schemes and an algebraic specification, produces a test set and the corresponding hypotheses.

Key Words: software testing, algebraic specifications, logic programming.

Introduction

Most the current methods and tools for software testing are based on the program to be tested: they use some coverage criteria of the structure of the program (control flow graph, data flow graph, call graph etc.). With the emergence of formal specification languages, it becomes possible to also start from the specification to define some testing strategies in a rigorous and formal framework. These strategies provide a formalization of the well known "black-box testing" approaches. They have the interesting property of being independent of the program; thus they result in test data sets which remain unchanged even when the program is modified. This is specially important for regression testing. Moreover, such strategies allow to test if all the cases mentioned in the specification are actually dealt with in the program. It is now generally agreed that "black-box" testing and "white-box" testing are complementary.

This paper presents both a theoretical framework for testing programs against formal specifications and a system, developed in Prolog, which allows to select a test data set from an algebraic specification and a testing strategy. All the "good" selection strategies, as defined in the theory, are supported by the system.

The starting point of the theoretical framework is the notion of an *ideal exhaustive test set* associated with the formal specification. The success of a program against this exhaustive test set is stated to be a correctness reference (under some hypotheses to be discussed later). Clearly, the definition of the exhaustive test set is determined by the semantics of the kind of formal specification which is used.

This exhaustive test set is of course not usable in practice since it is (most of the time) infinite and, moreover, not always decidable: there are some cases where it is not possible to decide whether or not an execution returns a correct result, mainly for reasons of observability. It is an aspect of the so called *oracle problem*. However, as said above,

the exhaustive test set provides a theoretical correctness reference, and we show how to *select*, by successive refinements, test sets which are finite, decidable and keep some other good properties of the exhaustive test set.

The crucial notion in the selection process is the concept of *testing hypotheses*, already introduced in [BCFG86] for different purposes. Hypotheses represent and formalize common test practices. Very roughly, when one chooses a finite test set $T = \{\tau_1, \ldots, \tau_n\}$ satisfying some criteria to test a property $P(x)$, one assumes at least the following hypothesis:

$$[P(\tau_1) \wedge P(\tau_2) \wedge \ldots \wedge P(\tau_n)] \implies \forall x, P(x)$$

As a matter of fact, such an hypothesis is the result of the combination of several simpler and more sensible hypotheses.
These hypotheses are usually left implicit. We believe that it is of first importance to make them explicit. Besides, we claim that they are a good conceptual tool for the selection of test data sets: it seems sound to state first some general hypotheses and then to select a test set corresponding to them. Actually, we propose to build these hypotheses by combination and specialization of some general hypothesis schemes.

Generally, stronger hypotheses will result in smaller test sets: the weakest hypotheses are the ones associated with the exhaustive test data set; the strongest one is that the program under test is correct and it corresponds to... an empty test set. The practically interesting pairs of hypotheses and test sets are obviously somewhere between these extremist views of testing. As usual in testing, the problem is to find a sound trade-off between cost and quality considerations.

As soon as specifications are handled in a formal framework and the testing strategies are expressed as hypotheses (i.e. formulas), one can consider the possibility of using proof-oriented tools for the selection of test data sets. These tools depend on the kind of formal specification in use: in this paper we present a tool based on "Horn clause logic" which allows to deal with algebraic specifications and produces test sets corresponding to the combination of some general hypothesis schemes. Our tool is even more elaborated since it helps in the choice of hypotheses.

Let us come back to the *oracle* problem, i.e. how to decide whether or not a program execution returns a correct result. The solutions to this problem depend both on the kind of formal specification and of the program: a property required by the specification may not be observable using the program under test. Most the formal specification methods provide a way to express observability. In this case, the program is assumed to satisfy the observability requirements (for instance to decide correctly the equality of two integers: it is an *oracle hypothesis*), and, following the same approach as above, an *exhaustive observable test set* can be associated with the specification. Then, testing hypotheses are used to select a finite subset of it. We present here such a solution in the framework of algebraic specifications.

Summing up, the theoretical framework presented here introduces the important idea that a test data set cannot be considered (or evaluated, or accepted, etc) independently of some hypotheses and of an oracle. Thus we define a *testing context* as a triple (H, T, O) where T is the test data set, H is a set of hypotheses and O an oracle. We then construct some basic testing contexts (roughly, those corresponding to the ideal exhaustive test sets

mentioned above) and we provide some way for deriving from them *practicable* testing contexts. Informally, a testing context (H, T, O) is practicable if: T is finite; O is defined on all the test data in T; it rejects no correct programs (*unbias*); and, assuming the hypotheses H, it accepts only correct programs (*validity*).

The paper is organized as follows:

- Section 1 introduces the general concepts and some theoretical results for a large class of formal specifications.
- Section 2 specializes the results of Section 1 to the case of structured algebraic specifications. It is shown that observability issues are crucial for the oracle problem. Besides, some fundamental hypotheses such as *uniformity hypothesis* or *regularity hypothesis* are introduced.
- Section 3 discusses the way these hypotheses can be combined.
- Section 4 presents the basic techniques which are used in the system. They are mainly extensions of equational logic programming
- Section 5 describes the system and the way it makes it possible to implement the strategies corresponding to the practicable testing contexts defined in Section 2. Moreover, the treatment of an example is reported.

1 Formal specifications and testing

1.1 Testing a program against its formal specification

In this first section, we consider that a specification is a set of formulas, written using a fixed set of logical connectors, and some operation names of a signature Σ. A program, which is supposed to implement the specification, provides a way of computing (for instance a function, a procedure) for each operation name of the signature Σ. These rather flexible definitions embeds various approaches of formal specifications such as temporal logic, algebraic specifications, etc[1].

Let *SP* be a formal specification and P be a program. It is possible to verify (by testing or by proving) the adequacy or inadequacy of P with respect to *SP* if the semantics of P and *SP* are expressible in some common framework. This role is ensured here by the concept of an interpretation for a given signature: intuitively a Σ-interpretation is a set of values plus, for each name in the signature Σ, an operation of the relevant arity on these values. We consider that the semantics of *SP* is a class of Σ-interpretations, and that P defines a Σ-interpretation. Then, the question of the correctness of P with respect to *SP* becomes: does the Σ-interpretation defined by P belong to the class of the interpretations of *SP*?

More precisely, a formal specification method is given by a *syntax* and a *semantics*.

[1] The readers familiar with Goguen and Burstall institutions will recognize a simplified version of them.

- The syntax is defined by a notion of *signature*. With each signature Σ is associated a set of *sentences* Φ_Σ. Φ_Σ contains all the well-formed formulas built on Σ, some variables, and some logical connectives.
- The semantics is defined by a class of Σ-*interpretations*, Int_Σ, and a *validation predicate* on $Int_\Sigma \times \Phi_\Sigma$ denoted by $\models$. For each Σ-interpretation A and for each formula ϕ, "$A \models \phi$" should be read as "A validates ϕ".

In this framework, a *formal specification* is a pair $SP = (\Sigma, Ax)$ such that Ax is a (finite) subset of Φ_Σ.

Notation 1 : the class of interpretations *validating SP* is called the class of *models* of *SP* and is denoted by $Mod(SP)$:

$$Mod(SP) \;=\; \{\; A \in Int_\Sigma \mid A \models Ax \;\}$$

The notions of signature, sentence, interpretation, and validation depend on the kind of formal specification, for instance algebraic specifications, temporal logic,...

What does it mean to test a program P against a Σ-formula $\phi(X)$?
As said above, $\phi(X)$ is a well-formed composition of logical connectives, operation names of Σ, and variables in X. Running a test of $\phi(X)$ consists of replacing the variables of X by some constants, computing by P the operations of Σ which occur in ϕ and checking that the results returned by P satisfy the property required by the connectives.
For instance, let *f, g, h, a, b* belong to Σ, let $\vee$ be a connective, *x,y* some variables; and $\phi(x,y)$ the following formula:

$$(\; f(x,y) \;=\; g(x)\;) \;\;\vee\;\; (\; f(x,y) \;=\; h(y)\;)$$

Let us note f_P, g_P, ...the functions computed by P for *f, g, ...*. A test data for the formula above is:

$$(\; f(a,b) \;=\; g(a)\;) \;\;\vee\;\; (\; f(a,b) \;=\; h(b)\;)$$

Running this test consists of computing the three values $f_P(a_P, b_P)$, $g_P(a_P)$, $h_P(b_P)$ and checking that the first one is equal either to the second one, or to the third one.
This view of program testing is just a generalization of the classical way of running tests, where the program is executed for a given input, and the result is accepted or rejected: in this case, the formula is the input-output relation required for the program.

We call a *test data set* of a Σ-formula $\phi(X)$ a set of instances of $\phi(X)$. As usual, when a finite test data set of reasonable size is successful, the program correctness is not ensured and when a test is not successful, we know that the program is not correct. As indicated in the introduction, this fact is expressed in our framework by hypotheses: given a formal specification *SP* and a program *P*, the test data selection problem consists of stating some hypotheses *H* and to select some test data set *T* such that:

$$H \;+\; Success(T) \;\;\Longrightarrow\;\; Correctness(P, SP).$$

When we get $Success(T)$ with an incorrect program, it means that the program does not meet the hypotheses H. The implication above is similar to the completeness criteria of Goodenough and Gehrart [GG75]: for every incorrect program that meets the hypotheses H, the test set T must fail.

In the discussion above, the oracle problem is hidden behind the notation *Success*. The oracle is some decision process, formalized by the predicate *Success*, which should be able to decide, for each elementary test τ in T if τ is successful or not when submitted to the program P. Providing such an oracle is not trivial at all ([Wey80][Wey82]) and may be impossible for some test data sets. Thus, the existence of *Success* must be taken into account, as well as the hypotheses H, when selecting a test data set T.

1.2 Testing contexts

Definition 2 : let P be the program under test and let $SP = (\Sigma, Ax)$ be its formal specification. A *testing context* is a triple *(H,T,Success)* where:

- H is a set of hypotheses about the interpretation A_P associated with P. (This means that H describes a subset $Mod_\Sigma(H)$ of Int_Σ: the set of Σ-interpretations "validating H")
- T is a subset of Φ_Σ; T is called a "test data set" and each element τ of T is called an "elementary test"
- the *Success* oracle is a partial predicate on Φ_Σ; for each formula (think "each elementary test") ϕ in Φ_Σ, either $Success(\phi)$ is undefined either it decides if ϕ is successful in A_P.

This definition is very general and calls for some comments.
Success can be shown as a procedure using the program P: one should write $Success_P$ (P and *SP* are implicit parameters of all the testing context).
It may seem surprising that test data sets can contain formulas with variables. Of course, our aim is to select test sets which are: executable (i.e. some set of ground formulas), finite, and as we will see later, instances of the axioms of *SP*. However, the definition above is useful because it allows to build testing contexts by refinements. A good starting point of what we call the "testing elaboration process" is the triple *("A_P is a Σ-interpretation", Ax_{SP}, undef)* where *undef* is the never defined predicate (as for test sets, more sensible oracles are built by refinements).
This initial testing context means that one wants to test the axioms of the specification, the oracle is not defined at all, and the hypothesis *"A_P is a Σ-interpretation"* simply means that each required functionality has been implemented (*not necessarily correctly* implemented). This hypothesis is equivalent to $Mod_\Sigma(H) = Int_\Sigma$.

The goal is to refine this initial testing context in order to get a practicable testing context. In the rest of this subsection, we give a sufficient condition to obtain practicability. Let us first formally define what is practicability.

Definition 3 : let *(H,T,Success)* be a testing context. Let us note $\mathcal{D}(Success)$ the definition domain of *Success*.

1. *(H,T,Success) has an oracle* means that:
   - T is finite[2]
   - If A_P validates H, then T is included in $\mathcal{D}(Success)$
2. *(H,T,Success)* is *practicable* means that:
   - *(H,T,Success)* has an oracle
   - If A_P validates H then $Success(T) \iff A_P \models Ax$ (where *Success(T)* denotes "$Success(\tau)$ for all τ in T"[3]).

We need to define some more properties on test data sets and oracles. These properties are sufficient conditions for practicability: "unbias" properties avoid rejection of correct programs; "validity" properties ensure that, assuming the hypothesis *H*, any incorrect program is discarded.

Definition 4 :

1. The test data set T is *valid* means that:
   If A_P validates H then

$$A_P \models T \implies A_P \models Ax_{SP}$$

2. The oracle *Success* is *valid* means that:
   If A_P validates H then

$$\forall \phi \in \mathcal{D}(Success),\ Success(\phi) \implies A_P \models \phi$$

3. The test data set T is *unbiased* means that:
   If A_P validates H then

$$A_P \models Ax \implies A_P \models T$$

4. The oracle *Success* is *unbiased* means that:
   If A_P validates H then

$$\forall \phi \in \mathcal{D}(Success),\ A_P \models \phi \implies Success(\phi)$$

A test data set is unbiased if and only if it contains only formulas which are theorems provable from the axioms of the specification. Equivalently, this means that a testing context should never require properties about the program which can discard correct programs.

The sufficient condition for practicability of testing contexts is given by the following fundamental fact.

Fact 5 : let *(H,T,Success)* be a testing context. If *(H,T,Success)* has an oracle and T and *Success* are both valid and unbiased, then *(H,T,Success)* is practicable.

[2] Actually, one should ask for T to be "of reasonable size"

[3] *Success(T)* is defined if *(H,T,Success)* has an oracle.

The proof is trivial. However, this fact is fundamental since it justifies the testing elaboration process: the initial testing context presented above is valid and unbiased; in the following subsection we give a refinement criterion which preserves validity; and in Section 2 we give another criterion, for algebraic specifications, which preserves unbias. Consequently, practicability is ensured providing that we stop the refinement process when the triple *(H,T,Success)* has an oracle.

1.3 The refinement preorder

In this subsection, we define the refinement preorder and we show how it makes it possible to build only valid testing contexts.

Definition 6 : let $TC_1 = (H_1, T_1, Success_1)$ and $TC_2 = (H_2, T_2, Success_2)$ be two testing contexts. TC_2 *refines* TC_1, denoted by $TC_1 \leq TC_2$, means that:

- "the hypotheses about the program under test may increase"

$$Mod_\Sigma(H_2) \subset Mod_\Sigma(H_1)$$

$$\text{or equivalently} \quad H_2 \implies H_1$$

- "under the hypotheses H_2, T_2 reveals as many errors as T_1 does"

$$\forall A_P \in Mod_\Sigma(H_2),\ \ A_P \models T_2 \implies A_P \models T_1$$

- "under the hypotheses H_2, the oracle $Success_2$ is more defined than, and consistent with $Success_1$"

$$\begin{gathered}\text{If } A_P \text{ validates } H_2 \text{ then}\\ \mathcal{D}(Success_1) \subset \mathcal{D}(Success_2) \text{ and}\\ \forall \phi \in \mathcal{D}(Success_1),\ \ Success_2(\phi) \implies Success_1(\phi)\end{gathered}$$

The refinement criterion is clearly a preorder[4] relation. Thus, it allows to built testing contexts incrementally (because of the transitivity). Let *(H,T,Success)* be a testing context; we have the following results.

Fact 7 : T is valid if

$$(\text{"}A_P \textit{ is a } \Sigma\textit{-interpretation"}, Ax_{SP}, undef) \leq (H,T,Success)$$

(The proof results directly from the definitions.) This fact is important because *("A_P is a Σ-interpretation",Ax_{SP},undef)* is the starting point of our testing context refinement process; thus, the validity of T is always ensured.

Fact 8 : *Success* is valid if

[4] it is not anti-symmetric

$$(\text{“}A_P \textit{ is a } \Sigma\textit{-interpretation”},\emptyset,[A_P \models _]_{|\mathcal{D}(Success)}) \leq (H,T,Success)$$

where $[A_P \models _]_{|\mathcal{D}(Success)}$ is the restriction of the formal validation predicate to the definition domain of *Success*.

(The proof results directly from the definitions.) We show in section 2 how to built oracles which trivially meet this property.

In this framework, the oracle is a decision procedure of the validity of the statement: $A_P \models \phi$. In most the cases, there is only a subset of the Σ-formulas ϕ such that this statement is practically decidable (see Section 2.3). We call *observable* those formulas such that $A_P \models \phi$ is trivially decidable for every program P. The oracle problem is then reduced either to select only observable tests (in that case the unbias property is trivial) either to extend observability by modifying the program under test, i.e. by adding procedures to P. However, this last solution is not always safe since the extensions may give biased results. In this paper we only consider the first solution. There are other possible approaches, for instance to require any program under test to be “fully observable”, i.e. to provide sufficient functionalities for deciding all the properties in its specification. This is related to the general problem of *design for testability* which obviously requires further research.

2 Algebraic specifications and testing

In the rest of this paper, we consider a special case of formal specifications: the theory of *algebraic abstract data types*. Section 2.1 recalls the main definitions; Section 2.2 shows that valid and unbiased test data sets can be selected from the so-called “exhaustive” test data set; Section 2.3 explains how the oracle problem can be handled and Section 2.4 proves that it is sufficient to select elementary tests reduced to a simpler form, namely ground equalities. This justifies that our system generates such simpler elementary tests selected from the exhaustive test data set.

2.1 Algebraic specifications

In the framework of algebraic specifications, a signature Σ is: a finite set S of *sorts* (i.e. type-names) and a finite set of *operation*-names with arity in S. The corresponding class of algebras Alg_Σ is defined as follows: a Σ-algebra is a heterogeneous set A partitioned as $A = \{A_s\}_{s \in S}$, and with, for each operation-name “$op : s_1 \times \ldots \times s_n \rightarrow s$” in Σ, a total function $op^A : A_{s_1} \times \ldots \times A_{s_n} \rightarrow A_s$. Σ-algebras correspond to the Σ-interpretations of Section 1. The formulas of Φ_Σ are positive conditional equations of the form:

$$(v_1 = w_1 \ \wedge \ldots \wedge \ v_k = w_k) \implies v = w$$

where v_i, w_i, v and w are Σ-terms with variables ($k \geq 0$).
An algebra A validates ($\models$) such a formula if and only if for each variable assignment σ with range in A, if $\sigma(v_i) = \sigma(w_i)$ for all i then $\sigma(v) = \sigma(w)$ (as in [ADJ76]).

Moreover, specifications are structured: a *specification module* ΔSP is a "part of specification" $(\Delta\Sigma, \Delta Ax)$ which uses some *imported* specifications $SP_i = (\Sigma_i, Ax_i)$ such that $SP = \Delta SP + (union\ of\ SP_i)$ is a (bigger) specification. The signature Σ of SP is the disjoint union of $\Delta\Sigma$ and the union of the Σ_i; the set of axioms Ax of SP is the union of ΔAx and the Ax_i. For example, a *List* specification over the imported specification of natural numbers can be expressed as in Figure 2.1.

1— A List Specification

$\Delta NATLIST$ uses NAT /* Here ΔNAT uses $BOOLEAN$ */

$\Delta S = \{ NatList \}$

$S_{Obs} = \{ Nat, Bool \}$

$\Delta\Sigma =$ $empty : \longrightarrow NatList$

$cons : Nat \times NatList \longrightarrow NatList$

$sorted : NatList \longrightarrow Boolean$

$insert : Nat \times NatList \longrightarrow NatList$

Generators: *empty, cons*

$\Delta Ax =$ $sorted(empty) = true$

$sorted(cons(N1, empty)) = true$

$sorted(cons(N1, cons(N2, L))) = and(\leq(N1, N2), sorted(cons(N2, L)))$

$insert(N1, empty) = cons(N1, empty)$

$\leq(N1, N2) = true \implies insert(N1, cons(N2, L)) = cons(N1, cons(N2, L))$

$\leq(N1, N2) = false \implies insert(N1, cons(N2, L)) = cons(N2, insert(N1, L))$

where $N1$ and $N2$ are variables of sort *Nat*, L of sort *NatList*.

Note that, given a structured specification *SP*, a subset S_{Obs} of *observable sorts* is distinguished among the sorts (*Nat* and *Bool* in the example). In our case, these observable sorts are specified by imported predefined specifications. They correspond to the observable primitive types of the used programming language; in particular for every program *P*, we assume that there are built-in equality predicates which are correct and defined on the observable sorts.
We also declare a set of *generators* included in $\Delta\Sigma$. This means that every element of an algebra validating the specification must be denotable by a ground term built on those generators. For example, every list must be denotable by a composition of *empty* and *cons*.
Lastly, let us remark that the *uses* mechanism is transitive: if A uses B and B uses C then A uses C. This transitive view of imports is not the most common one in programming languages, but it is the most common one in specification languages.

2.2 The exhaustive test set

Of course, all the results of Section 1 remain for the particular case of structured algebraic specifications. In particular, given a testing context *(H,T,Success)*, an elementary test of *T* can be any positive conditional equation. This subsection proves that it is possible to

select only ground instances of them without loosing validity. Moreover, in that case, we show that the unbias property is easily obtained.

Definition 9 : given a specification ΔSP, the *exhaustive* test data set, $Exhaust_{SP}$, is the set of all ground instances of all the axioms of ΔSP.

$$Exhaust_{SP} \;=\; \{\sigma(\phi) \mid \phi \in \Delta Ax \;,\; range(\sigma) = W_\Sigma\}$$

where W_Σ is the set of ground terms on Σ.

Our first result about $Exhaust_{SP}$ is that it allows to select test data sets which are never biased. Remember that it was difficult to practically ensure this property in the general case of formal specifications (Section 1).

Fact 10 : let *(H,T,Success)* be a testing context. If $T \;\subset\; Exhaust_{SP}$ then T is unbiased.

(This fact results directly from the definitions.)

The Σ-*adequacy hypothesis* defined below means that the program under test only exports the specified operations (i.e. there are no exported operations which are not specified). Under this non restrictive hypothesis $Exhaust_{SP}$ always produces valid test data sets via testing context refinements.

Definition 11 : let A_P be the algebra associated with P.

$$Adequat_\Sigma(A_P) \iff A_P \textit{ is a finitely generated } \Sigma\textit{-algebra}$$

(a Σ-algebra A is finitely generated if every value of A is denotable by a ground Σ-term.)

Fact 12 : let *(H,T,Success)* be a testing context.
If $(H, T, Success) \;\geq\; (Adequat_\Sigma, Exhaust_{SP}, undef)$ then T is valid.

This fact results from:

$$(\text{“}A_P \textit{ is a } \Sigma\textit{-algebra”}, Ax_{\Delta SP}, undef) \leq (Adequat_\Sigma, Exhaust_{SP}, undef)$$

and from the validity fact of Section 1.3.

In particular the exhaustive test data set is valid; but unfortunately it is generally infinite. The previous facts mean that the testing context refinement process can be reduced to “add hypotheses in order to select a finite (pertinent) subset of $Exhaust_{SP}$.”

Let us show on an example what kind of hypotheses can be used. Let us treat the axiom

$$\leq(N1, N2) = true \implies insert(N1, cons(N2, L)) = cons(N1, cons(N2, L))$$

One has to select instances of the list variable L and instances of the natural number variables $N1$ and $N2$. A first idea can be to bound the size of the list terms substituting L. This can be obtained via the general schema of *regularity hypothesis*.

Regularity hypothesis: let $\phi(L)$ be a formula involving a variable L of a sort $s \in \Delta S$. Let $|t|_s$ be a complexity measure on the terms t of sort s (for instance the number of operations of sort $s \in \Delta\Sigma$ occurring in t). Let k be a positive integer. A regularity hypothesis of level k, $Regul_{\phi,k}(A_P)$, is expressed as follows:

$$(\forall t \in W_\Sigma)(\ |t|_s \leq k \ \Rightarrow\ A_P \models \phi(t)\ ) \quad \Longrightarrow \quad (\forall t \in W_\Sigma)(\ A_P \models \phi(t)\ )$$

where W_Σ is the set of all ground Σ-terms.

For example, a regularity level 3 allows to select the following instances of L:

$L = empty$
$L = cons(N3, empty)$
$L = insert(N3, empty)$
$L = cons(N4, cons(N3, empty))$
$L = insert(N4, cons(N3, empty))$
$L = cons(N4, insert(N3, empty))$
$L = insert(N4, insert(N3, empty))$

The corresponding refinement of the testing context is obtained by adding the regularity hypothesis of level 3 to H, and by replacing the previous axiom by the seven axioms related to those seven instances of L. Consequently, only variables of sort natural number remain: $N1, N2, N3$ and $N4$.

The four instances of L involving the operation *insert* seem to be less relevant than those involving only *cons* and *empty* because *cons* and *empty* generate all the list values. Let us note $\Delta\Omega$ the set of generators. The operations of $\Delta\Sigma - \Delta\Omega$, such as *insert*, are called *defined operations*. Here, the treated axiom is one of the three *defining axioms* of *insert* in *NATLIST*. Instances of L involving only generators of the sort s of L can be selected via the so called Ω-regularity hypothesis.

Ω-Regularity hypothesis: let $W_{\Delta\Omega+\Sigma_1\ldots+\Sigma_n}$ be the set of the Σ-terms which do not contain defined operations (Σ_i are the imported signatures). Let $|t|_s$ be a complexity measure defined on those terms of sort s. A Ω-regularity hypothesis of level k is expressed as follows:

$$(\forall t \in W_{\Delta\Omega+\Sigma_1\ldots+\Sigma_n})(\ |t|_s \leq k \ \Rightarrow\ A_P \models \phi(t)\ ) \quad \Longrightarrow \quad (\forall t \in W_\Sigma)(\ A_P \models \phi(t)\ )$$

Then, a regularity level 3 allows to select only the three interesting instances of L. In our system, we mainly use a Ω-regularity hypothesis for each sort of ΔS and the user can choose the level k.

At this stage in the example, the test selection problem is reduced to the replacement of the variables of sort natural number by ground terms. In the general case, after a finite number of regularity hypotheses, only variables belonging to imported sorts remain. The replacement of these variables by ground terms can be obtained "by brute force" using *uniformity hypothesis*.

Uniformity hypothesis: let $\phi(V)$ be a formula involving a variable V of imported sort s. A uniformity hypothesis, $Unif_{\phi}(A_P)$, is expressed as follows:

$$(\forall v_0 \in W_{\Omega,s})(\ A_P \models \phi(v_0) \implies [\forall v \in W_{\Sigma,s}][A_P \models \phi(v)]\ )$$

Such an hypothesis means that if a formula is true for some value v_0 then it is always true ... This is a strong hypothesis, and it may seem unreasonable at first glance. However, it states explicitly what is often assumed when testing a program.

Of course, a uniformity hypothesis should not be applied directly to conditional axioms because one has few chance to validate the precondition of the axiom. For example, the formula

$$\leq(N1, N2) = true \implies insert(N1, cons(N2, empty)) = cons(N1, cons(N2, empty))$$

could becomes

$$\leq(2, 0) = true \implies insert(2, cons(0, empty)) = cons(2, cons(0, empty))$$

which is not relevant since $\leq(2, 0)$ is false. Thus, uniformity hypothesis must be used carefully. We often use uniformity on appropriate *subdomains*. We show in Section 3 how we automatically derive these uniformity subdomains.

The modularity of the specification is crucial here. The choice of the hypotheses is guided by the specification structure: the regularity hypotheses are made for the sorts specified by ΔSP while the uniformity hypotheses are made for the imported sorts.

2.3 The oracle

In this subsection, we show how the oracle problem can be handled using observability issues. Some more elaborated solutions are described in [Ber89].

Assuming that a finite test data set T has been selected from $Exhaust_{SP}$, an elementary test is of the form:

$$(t_1 = u_1 \wedge \ldots \wedge t_k = u_k) \implies t = u$$

where t_i, u_i, t and u are ground Σ-terms. The oracle problem is reduced to decide success/failure of *equalities* between ground terms, because the truth tables of $\wedge$ and $\implies$ can then be used to decide success/failure of the whole conditional axiom. As already pointed out in [BCFG86], such a decision is not always trivial. For example, let P implements stacks by arrays: *push* records the element at range *height* and increments *height*, *pop* simply decreases *height*. Suppose the oracle has to decide if $pop(push(3, empty))$ and *empty* give the same stack after their executions via P. There is an observability problem which, indeed, is a concrete equality problem: these stacks get two distinct array representations (because 3 has been recorded in the array for the first stack) but they are abstractly equal (as the common height is 0). Thus $Success(pop(push(3, empty)) = empty)$ must be decided via some carefully elaborated method.

We may add new procedures to the program, extracting the concrete representations and computing the abstract equalities. However the added procedures may alter the program behaviour, they must be proved or tested; moreover, if they are added into the program code, some of the advantages of black-box testing are lost. A better solution could be to replace each test *[t=u]* by a set of tests of the form *[C(t)=C(u)]* obtained by surrounding t and u with some well chosen *observable contexts C*.

For the stack example, assuming that integers and elements are observable, $t{=}u$ seems to be equivalent to the observable following equalities:

$$[height(t) = height(u)]$$

$$[top(t) = top(u)]$$

$$[top(pop(t)) = top(pop(u))]$$

$$\dots$$

$$[top(pop^{height-1(t)}(t)) = top(pop^{height-1(t)}(u))]$$

Here the observable contexts are $height(_)$ and the $top(pop^i(_))$ such that $0 \leq i < height(t)$. Unfortunately identifying such a minimal set of contexts is undecidable in the general case (see [Sch86], page 8). Besides, when testing "big" stacks, this leads to an impracticable number of observable tests.

But the situation is even worst: when we think that $height(_)$ and $top(pop^i(_))$ are sufficient, we implicitly assume that we manipulate (more or less) stacks. But this fact is just what we check when testing !

Counter-example: we sketch here a program which does not validate this implicit hypothesis. The "bug" is that *top* returns the height when applied to a term t of the particular form $t = push(e, pop(\dots))$; for every other term t, $top(t)$ returns the correct value. A "stack" is implemented by a record `<array,height,foo>` where `<array,height>` is the usual correct implementation and `foo` records the number of `push` performed since the last `pop` (and the empty stack is initialized with `foo=2`). [We ignore the exceptional cases of `pop` and `top` when the stack is empty, it is not our purpose here...].

```
proc emptystack();
    stack.height := 0 ; stack.foo := 2 ;;

proc push(e:element);
    stack.array[stack.height] := e ;
    stack.height := stack.height+1 ; stack.foo := stack.foo+1 ;;

proc pop();
    stack.height := stack.height - 1 ; stack.foo := 0 ;;

proc top();
    if (stack.foo = 1) then return stack.height   /* the bug */
    else return stack.array[stack.height] ;;
```

The terms $t = push(1, emptystack)$ and $u = pop(push(2, push(1, emptystack)))$ are distinguishable because we get `top(push(0,t))=0` (as `foo=4`) and we get `top(push(0,u))=2` (as `foo=1`). Nevertheless, the contexts $height(x)$ and $top(pop^i(x))$ are unable to distinguish t from u (as `foo` is never equal to 1 for those contexts), leading to an oracle which never detects that $t \neq u$.

So, we get the depressing result that identifying the minimal set of observable contexts for the equality decision *is decidable* ! We must consider the set of *all* observable contexts... which is infinite (consequently impracticable).

A first idea is to follow a similar approach as for test selection: we may add oracle hypotheses in order to reduce this infinite set of contexts to a finite one, SC, with informally:

Oracle hypotheses and success(SC) $\Longrightarrow$ success of the original elementary test

Remark 13 : let us consider the following oracle hypothesis:
"*for all stacks t and u, if height(t)=height(u) and* $top(pop^i(t)) = top(pop^i(u))$ *for i=0..height(t)-1 then t=u*"
This hypothesis makes it posssible to reduce the infinite set of all contexts to the finite usual one: $height(_)$ and $top(pop^i(_))$.

The advantage of our approach is that the oracle hypothesis is explicitly mentioned. Moreover, for "big" stacks, a more powerful hypothesis can allow to select only a subset of all these *top*s. In such cases, the counter-example given above does not meet the oracle hypothesis and our black-box approach does not reveal the bug. Notice that any complementary white-box testing would find it (see the introduction).

The idea of adding oracle hypotheses works when deciding equalities which appear *in the conclusion* of the tested conditional axioms. If the program under test does not meet the oracle hypothesis then the oracle may accept wrong results, but this permissivity is just expressed by the hypothesis. Thus the validity of the testing context is not lost.
Unfortunately, if the equality appears in the precondition, for instance

$$t = u \quad \Longrightarrow \quad concl\ ,$$

then one get a biased oracle! This is due to the fact that $t=u$ may be successful according to the oracle hypotheses, but not valid. In that case one would require for *concl* to be true, in spite of the fact that *concl* is not required according to the formal validation predicate.

Fortunately a solution exists when the preconditions of all the axioms of SP belong to observable sorts only (which is the case for most specifications). Let us recall (Section 2.1) that there is a subset $S_{Obs} \subset S$ with implemented built-in correct equality predicates, denoted by $\{eq_s\}_{s \in S_{Obs}}$.

We first define the "minimal hypothesis" which formalizes the properties of the observable sorts, then we define the exhaustive observable test data set Obs_{SP}. Finally, we show that this data set allows to produce valid and unbiased oracles, in a similar way as $Exhaust_{SP}$ does for test data set without oracle.

Definition 14 : the "minimal hypothesis", denoted $Mini(A_P)$, is the conjunction of $Adequat_\Sigma(A_P)$ and the following:
for any ground equality $t{=}u$, if the sort s of t and u belongs to S_{Obs} then

$$A_P \models t = u \iff A_P \models eq_s(t, u) \iff SP \vdash t = u$$

(where "$\vdash$" stands for equational reasoning and structural induction[5]) else

$$A_P \models t = u \iff A_P \models eq_s(C(t), C(u)) \text{ for all observable contexts } C$$

This minimal hypothesis reflects the classical notion of correctness up to *behavioural equivalence* (as defined in [Kam83][ST87] or [Hen91] for instance).

In the rest of this paper we assume that SP contains only axioms of the form $(L \implies R)$ where L is observable (or empty). Thus, the exhaustive test data set $Exhaust_{SP}$ contains elementary tests of the form $(L \implies t = u)$ where L contains only ground equalities of observable sorts. Of course, the problem of bias mentioned before does not remain since the built-in observable equality predicates are correct (from *Mini*).

Definition 15 : the *exhaustive observable test data set* is the set Obs_{SP} whose elements are the ground formulas $[L \implies C(t) = C(u)]$ such that $(L \implies t = u)$ is element of $Exhaust_{SP}$ and C is any observable context over the sort of $t{=}u$.

The following definition and results show that one can reach practicability by selecting a finite subset of Obs_{SP}. This gives a solution to our problem.

Definition 16 : $Success_{Obs}$ is the oracle defined as follows:
$Success_{Obs}$ coincides with the implemented built-in predicates eq_s for all ground equalities of observable sort $s \in S_{Obs}$; $Success_{Obs}$ uses the truth tables of $\wedge$ and $\implies$ in a straightforward manner for conditional axioms involving only observable ground equalities; and $Success_{Obs}$ is undefined otherwise.

Lemma 17 : for any testing context of the form $(H, T, Success_{Obs})$ such that $H \implies Mini$, the oracle $Success_{Obs}$ is valid and unbiased.

This results from the facts that the truth tables of $\implies$ and $\wedge$ are correct and complete with respect to the validation predicate and that the hypothesis *Mini* implies that each eq_s is valid, unbiased and defined for all observable ground equalities.

Lemma 18 :

$$(Adequat_\Sigma, Exhaust_{SP}, undef) \le (Mini, Obs_{SP}, undef)$$

In particular Obs_{SP} is a valid test data set under the hypothesis *Mini*.

(The proof results directly from the definitions)

[5] Note that equational reasoning and structural induction is correct and complete for *ground* equations

Fact 19 : if *(H,T,Success)* is a testing context such that:

- $(H,T,Success) \geq (Mini, Obs_{SP}, Success_{Obs})$
- $T \subset Obs_{SP}$ and T is finite
- $Success = Success_{Obs}$,

then *(H,T,Success)* is practicable.

Proof : from the last fact of Section 1.2 it is sufficient to show that *(H,T,Success)* has an oracle, T and *Success* are unbiased, T and *Success* are valid. Since *Obs* involves only observable ground conditional equalities and *Mini* implies that eq_s is defined on every observable ground equality, the fact that H is stronger than *Mini* implies that the definition domain of $Success_{Obs}$ contains Obs_{SP}. Consequently since T is a finite subset of Obs_{SP}, $(H,T,Success_{Obs})$ has an oracle. Since $Exhaust_{SP}$ is unbiased and Obs_{SP} is built from $Exhaust_{SP}$ by adding contexts, Obs_{SP} remain unbiased. The validity of T results from the inequality given in the previous fact and from the first fact of Section 1.3 Moreover $Success_{Obs}$ is unbiased and valid from the previous fact. □

From the theoretical point of view, the previous fact is fundamental because it gives useful sufficient conditions to solve the test selection and oracle problems. From now on, one can consider only elementary tests of the form $[L \implies R]$ where all the equalities occurring in L and R are observable (consequently decidable). Our last theoretical improvement is to show that one can skip the elementary tests such that L is not satisfied.

2.4 Reducing test to equalities

Let us consider a test of the form $[L \implies R]$ such that L contains only observable ground equalities and R is a ground equality. Let us assume that L is not satisfied by the program under test. From the truth table of $\implies$, $[false \implies R]$ is always true. It follows that the elementary test $[L \implies R]$ has no chance to reveal any error of the program; thus it is (at least intuitively) useless (see Section 1.1). Consequently it is of first interest to select only elementary tests of the form $[L \implies R]$ such that L is true. This means that, given a conditional axiom of ΔSP

$$(v_1 = w_1 \wedge \ldots \wedge v_k = w_k) \implies v = w \ ,$$

one should select instances of the variables which satisfy the precondition ($\bigwedge_{i=1..k} v_i = w_i$). It is the reason why an equational resolution procedure *a la* PROLOG is needed for automatizing test selection (see sections 4 and 5).

Now, let us assume that the precondition L is true. From the truth table of $\implies$, $[true \implies R]$ is always equivalent to R alone. It follows that submitting $[L \implies R]$ is useless, it is sufficient to submit R to the program under test. Consequently, assuming that the specification allows to decide whether the precondition is true or false, our system can select only instances of

$$(v_1 = w_1 \wedge \ldots \wedge v_k = w_k) \implies v = w \ ,$$

satisfying ($v_1 = w_1 \wedge \ldots \wedge v_k = w_k$) and produce only the related instances of $v{=}w$.

The only difficulty is that "L is true with respect to the specification" is *a priori* not equivalent to "L is true with respect to the program." Indeed, the following definition and results prove that there is no problem.

Definition 20 : The *exhaustive equational test data set* is the set $EqExh_{SP}$ whose elements are the ground equalities *[t=u]* such that:
($t_1 = u_1 \wedge \ldots \wedge t_k = u_k \implies t = u$) is element of $Exhaust_{SP}$ and each equality $t_i = u_i$ is provable from the specification (using equational reasoning and structural induction).

Fact 21 : let *(H,T,Success)* be a testing context. If $T \subset EqExh_{SP}$ then T is unbiased.

Proof : let us remind that if T only contains theorems of *SP* then it is unbiased (Section 1.2). By definition, $EqExh_{SP}$ only contains theorems of the axioms of *SP* (because equational reasoning and structural induction is correct), which implies the fact. □

Fact 22 : let *(H,T,Success)* be a testing context. Let us assume that ΔSP contains only axioms whose preconditions are observable.
If $(H, T, Success) \ \geq \ (Mini, EqExh_{SP}, undef)$ then T is valid.

Proof : from the validity fact of Section 2.2 and by transitivity of "$\geq$", it is sufficient to prove that:

$$(Mini, EqExh_{SP}, undef) \ \geq \ (Adequat_{\Sigma}, Exhaust_{SP}, undef)$$

Thus it is sufficient to prove:

$$\forall\, A_P \in Mod(Mini) \ , \quad A_P \models EqExh_{SP} \implies A_P \models Exhaust_{SP}$$

Let us assume that A_P validates *Mini* and that $A_P \models EqExh_{SP}$. Let τ :

$$(t_1 = u_1 \wedge \ldots \wedge t_k = u_k \implies t = u)$$

be an elementary test of $Exhaust_{SP}$; one has to prove that $A_P \models \tau$. Since A_P validates *Mini* and $t_i = u_i$ is observable for every $i = 1..k$, two cases are possible:

- $A_P \models eq_{s_i}(t_i, u_i)$ for every $i = 1..k$. In that case, $[t = u]$ belongs to $EqExh_{SP}$ because $SP \vdash t_i = u_i$ from the *Mini* hypothesis. Consequently, since $A_P \models EqExh_{SP}$, $A_P \models t = u$, which implies that $A_P \models \tau$.
- there exists i such that A_P does not validate $eq_{s_i}(t_i, u_i)$. In that case, A_P does not validate $t_i = u_i$ (from the *Mini* hypothesis). Consequently, $A_P \models \tau$ (as the precondition of τ is not true).

□

The two previous facts prove that it is sufficient to submit equational elementary test instead of conditional ones. In fact it is possible to select only equational observable elementary tests, as stated below:

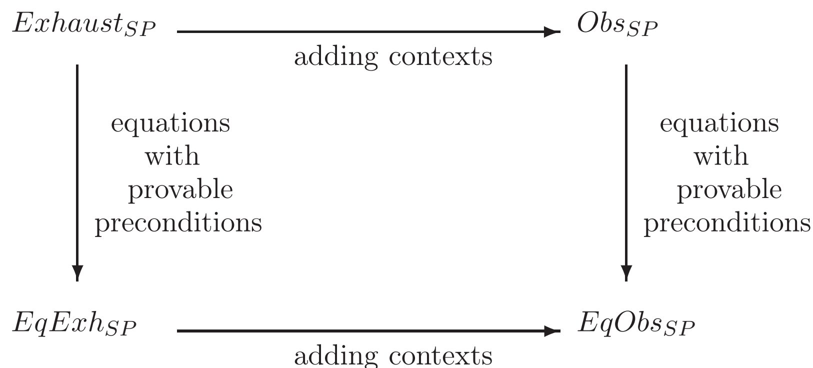
Definition 23 : the *exhaustive observable equational test data set* is the test data set $EqObs_{SP}$ whose elements are the ground equalities *[t=u]* such that: $(t_1 = u_1 \wedge \ldots \wedge t_k = u_k \implies t = u)$ is element of Obs_{SP} and each equality $t_i = u_i$ is a theorem of the specification (using equational reasoning and structural induction).

Fact 24 : let us assume that ΔSP contains only axioms whose preconditions are observable. If *(H,T,Success)* is a testing context such that:

- $(H, T, Success) \geq (Mini, EqObs_{SP}, \{eq_s\}_{s \in S_{Obs}})$
- $T \subset EqObs_{SP}$ and T is finite
- $Success = \{eq_s\}_{s \in S_{Obs}}$

then *(H,T,Success)* is practicable.

Proof : $EqObs_{SP}$ can be also obtained from $EqExh_{SP}$ as follows: the elements of $EqObs_{SP}$ are the ground equations of the form $C(t) = C(u)$ such that $t = u$ is element of $EqExh_{SP}$.



The previous fact can be proved in a similar way as the fact of Section 2.3 above (by replacing $Exhaust_{SP}$ and Obs_{SP} by $EqExh_{SP}$ and $EqObs_{SP}$ respectively). □

These results provide some guidelines for the selection of test data sets for a specification module ΔSP: the problem is to select a finite subset of $EqObs_{SP}$. As we already said, the reduction of $EqObs_{SP}$ to a finite subset is obtained by stating regularity (or Ω-regularity) hypotheses on the defined sorts, uniformity hypotheses on the imported sorts and oracle hypotheses for the observable contexts. The previous fact proves that the resulting testing contexts are practicable. However, we have seen that the composition of these hypotheses must be done carefully: we gave an example with a conditional axiom in Section 2.2. Indeed, we are faced to a more general problem which is discussed below.

3 Choice of regularity and uniformity hypotheses

3.1 The problem

Putting together uniformity hypotheses and regularity hypotheses is not straightforward. Brute uniformity hypotheses turn out to be too strong, in some cases for equations and in most the cases for conditional axioms. By too strong, we mean that some obviously relevant cases are not tested under these hypotheses.

An instance of the problem for a conditional axiom has been given in section 2.2: applying uniformity hypotheses on some sorts consists of assigning arbitrary values to the variables of these sorts ; this can result in an instance of the axiom where the precondition is false ; thus testing this instance is meaningless. In this case, the uniformity hypotheses must not be made on the whole domains of the variables, but on a subdomain which is the validity domain of the precondition.

The notion of uniformity subdomain is not only useful for conditional axioms, but also, even if it is less obvious, for equations. Let us consider the following equation of the specification of lists of natural numbers :

$$sorted(cons(N1, cons(N2, L))) = and(\ \leq(N1, N2), sorted(cons(N2, L)))$$

Ω-regularity hypothesis of level 2 on lists yields the following set of instances:

1. $sorted(cons(N1, cons(N2, empty))) =$
   $and(\leq(N1, N2), sorted(cons(N2, empty)))$
2. $sorted(cons(N1, cons(N2, cons(N3, empty)))) =$
   $and(\leq(N1, N2), sorted(cons(N2, cons(N3, empty))))$

A uniformity hypothesis on natural numbers would result in two ground equations (the remaining *Nat* variables are replaced by some ground terms). It has good chance to result in a test data set where some interesting cases in the definition are not covered. For instance:

$$sorted(cons(5, cons(3, empty))) = and(\ \leq(5, 3), sorted(cons(3, empty)))$$

$sorted(cons(4, cons(2, cons(6, empty)))) =$
$and(\ \leq(4, 2), sorted(cons(2, cons(6, empty))))$

Indeed, a better coverage should at least reach the following cases:

- $\leq(N1, N2) = true\ \wedge\ sorted(N2, L) = true$
- $\leq(N1, N2) = true\ \wedge\ sorted(N2, L) = false$
- $\leq(N1, N2) = false\ \wedge\ sorted(N2, L) = true$
- $\leq(N1, N2) = false\ \wedge\ sorted(N2, L) = false$

(In the test data set above, only the third case is covered). These four cases define four subdomains for the tested axiom. These subdomains come from the decomposition of the definition of *sorted* and from the properties of *and*. Uniformity hypotheses on these subdomains can be made; in that case, they are called *uniformity subdomains*. However, the decompositions could be continued using the axioms of $\leq$ or the defining axioms of *sorted* (including the one under test).

3.2 Decomposition of subdomains by unfolding defined operations

What we have done above is a decomposition of the domain of *sorted*. In this subsection we describe how to decompose defined operations in a systematic way, in order to get relevant uniformity subdomains for the axioms where they occur[6].

As usual in testing methods, this decomposition is based on case analysis: the preconditions which occur during the decomposition are composed by conjunction. This is done until a satisfactory coverage of the cases mentioned in the specification is obtained. Of course, the level of case coverage is strongly correlated to the size of the data set. It is not always possible or realistic to push the decomposition too far. At the end of this section, we give some hints on how to stop the decomposition with relevant associated uniformity hypotheses.

In order to describe the decomposition of defined operations in a legible way, we assume that the specification has the following form: for each defined operation f there is a set of defining axioms where f is the top symbol of the left hand side of the conclusion. Thus a defining axiom looks like:

$$precondition \Rightarrow f(t_1, \ldots, t_n) = t$$

where *precondition* may be either empty or a conjunction of equations. We assume that the left hand-side of the conclusion is defined by the right hand-side. Thus, axioms are implicitly oriented from left to right.

Let us assume that we want to find subdomains for a defining axiom of f. Let $\phi_f = (precond_f \Rightarrow concl_f)$ be this axiom. Suppose that there is an occurrence u of a defined operation g in ϕ_f (which is not the top of the left hand-side of $concl_f$). Let $\phi_g = (precond_g \Rightarrow g(t_1, \ldots, t_m) = t)$ be one of the axioms defining g. Let $g(a_1, \ldots, a_m)$ be the subterm at the occurrence u in ϕ_f. The unfolding of ϕ_f via ϕ_g at the occurrence u can be described in two steps:

- we first replace $g(a_1, \ldots, a_m)$ by t in ϕ_f at the occurrence u.
  Let $(precond' \Rightarrow concl')$ be the resulting formula.
- then we add into $precond'$ the equalities between the related arguments of g and the precondition of ϕ_g. Thus, the unfolded axiom is the following one:

$$precond' \ \wedge \ t_1 = a_1 \ \wedge \ldots \wedge \ t_m = a_m \ \wedge \ precond_g \Rightarrow concl')$$

[6]The result of these decompositions is a partition of the domain of the axiom where the operations occur into subdomains. The original axiom, which is unchanged, will be tested once for each subdomain.

This unfolding is done at occurrence u with all axioms defining g, giving as many unfolded axioms as there are axioms defining g.

It appears that for testing ϕ_f we have the choice between:

- replacing directly its variables by ground terms which satisfy $precond_f$ (without unfolding), then the corresponding hypothesis is uniformity on the validity domain of $precond_f$
- performing a similar replacement for each of the unfolded axioms. Of course, the test data set is larger. We get more, but weaker, uniformity hypotheses.
- if there is still some occurrence of a defined operation in one of the unfolded axioms, continuing the subdomains decomposition by unfolding (before replacing).

When g is f itself (recursive definition of f), this transformation is similar to the classical Burstall and Darlington's unfolding of recursive definitions.

It is clear that in most the cases, such a decomposition is infinite. However, assuming some regularity and uniformity hypotheses (on subdomains), it is possible to make these decompositions finite, as seen in the example below.

Let us come back to our axiom on *sorted* (section 2.1), and assume that the defining axioms of *and* and $\leq$ are:

$$\begin{array}{ll}
and(true, true) = true & \leq(N1, N1) = true \\
and(true, false) = false & <(N1, N2) = true \ \Rightarrow \leq(N1, N2) = true \\
and(false, true) = false & <(N2, N1) = true \ \Rightarrow \leq(N1, N2) = false \\
and(false, false) = false &
\end{array}$$

where $<$ is inductively defined on the natural numbers generated, as usual, by zero and successor. For legibility purpose, these defining axioms explicitly reflects the cases that we want to cover in the decompositions.

As seen above, a regularity hypothesis of level 2 gives the two following instances:

1. $sorted(cons(N1, cons(N2, empty))) =$
   $and(\ \leq(N1, N2), sorted(cons(N2, empty)))$
2. $sorted(cons(N1, cons(N2, cons(N3, empty)))) =$
   $and(\ \leq(N1, N2), sorted(cons(N2, cons(N3, empty))))$

Let us consider the first equation.

1. From the first axiom of $\leq$, it comes:

   $N1 = N2 \Longrightarrow$
   $sorted(cons(N1, cons(N2, empty))) = and(true, sorted(cons(N2, empty)))$

From the definition of *and*, we get two cases:

- the recursive call of *sorted* is true

  $$N1 = N2 \ \wedge \ sorted(cons(N2, empty)) = true \implies sorted(cons(N1, cons(N2, empty))) = true$$

  and from the axioms of *sorted* we get $true = true$ in the precondition, which can be suppressed (indeed, there are more cases, but each of them lead to an unsatisfiability in the precondition).
  A uniformity hypothesis on the domain of $N1 = N2$ seems sensible, thus we stop the decomposition.

- if the recursive call of *sorted* is false, we get $true = false$ in the precondition, thus we ignore this irrelevant case.

2. From the second axiom of $\leq$, it comes:

   $$<(N1, N2) = true \implies sorted(cons(N1, cons(N2, empty))) = and(true, sorted(cons(N2, empty)))$$

   From the axioms of *and*, we get only one case (following the same method as in 1).

   $$<(N1, N2) = true \ \wedge \ sorted(cons(N2, empty)) = true \implies sorted(cons(N1, cons(N2, empty))) = true$$

   and from the axioms of *sorted* we can eliminate $sorted(cons(N2, empty)) = true$. A uniformity hypothesis on the domain of $<(N1, N2) = true$ seems sensible, thus we can stop the decomposition.

3. From the last axiom of $\leq$, it comes:

   $$<(N2, N1) = true \implies sorted(cons(N1, cons(N2, empty))) = and(false, sorted(cons(N2, empty)))$$

   From the axioms of *and*, we get two cases:

   - the recursive call of *sorted* is true

     $$<(N2, N1) = true \ \wedge \ sorted(cons(N2, empty)) = true \implies sorted(cons(N1, cons(N2, empty))) = false$$

     and from the axioms of *sorted* we can eliminate the second equality of the precondition.
     As in the previous case, a uniformity hypothesis on the domain of the precondition seems sensible, thus we stop the decomposition.

   - if the recursive call of *sorted* is false, we get a contradiction with the axioms defining *sorted*, thus we ignore this case.

All the preconditions of these cases define the coverage we want for the first instance of the axiom. They define the uniformity subdomains of the domain of the axiom under test (in this case, the natural numbers, since the axiom is not conditional) as follows:

- $N1 = N2$

- $<(N1, N2) = true$
- $<(N2, N1) = true$

Thus, for the first instance of the axiom generated by regularity, we select three instances (one for each uniformity subdomain).

For the second instance of the axiom generated by regularity, we follow the same way and get nine decomposition subdomains defined by:

- $N1 = N2 \ \wedge \ N2 = N3$
- $N1 = N2 \ \wedge \ <(N2, N3) = true$
- $N1 = N2 \ \wedge \ <(N3, N2) = true$
- $<(N1, N2) = true \ \wedge \ N2 = N3$
- $<(N1, N2) = true \wedge \ <(N2, N3) = true$
- $<(N1, N2) = true \wedge \ <(N3, N2) = true$
- $<(N2, N1) = true \ \wedge \ N2 = N3$
- $<(N2, N1) = true \wedge \ <(N2, N3) = true$
- $<(N2, N1) = true \wedge \ <(N3, N2) = true$

Clearly, the selected uniformity subdomains depends both of the axioms of the specification and of the criteria we choose to stop the decompositions. This is not surprising since any black-box testing method depends on the specification. We present in the next section a tool which enables us to stop such decompositions using some control specifications. This tool provides a mean to automatically compute uniformity subdomains for each axiom.

4 Basic tools for automatizing test data selection

This section presents in a detailed way the principles and internal mechanisms of a system which makes it possible to automatize the approach presented in Sections 2 and 3. This system takes as input an algebraic specification and some indications on regularity hypotheses and halting of unfolding. It provides uniformity subdomains and the corresponding test data. The reader which is not familiar with logic programming techniques can skip this section and go to section 5 where an example of the use of the system is given. Section 5 refers to Section 4 but is understandable by itself.

Two of the main difficulties in automatizing the selection are:

1. the computation of the elements of a uniformity subdomain

2. the decomposition, using the axioms, of a domain into uniformity subdomains.

We present here a procedure and some control strategies which allow to cope with these difficulties.

To solve the first point, we need an equational resolution procedure to compute the solutions of the equations that define a uniformity subdomain, which of course must be correct (any returned value is a solution) and complete (all the solutions are computed). In the case of positive conditional axioms, there exist such procedures: the conditional narrowing presented in [Hus85] for the RAP language; the clausal superposition for equational Horn clauses presented in [Fri85] for the SLOG language.

In a first step, we performed several experiments for automatizing the test data selection using RAP and SLOG. These experiments allowed us to study and to identify the specific control mechanisms for the implementation of our selection strategies. The introduction of these control mechanisms in either RAP or SLOG implied some important changes in the interpreters. Thus, we chose to use a general language to simulate equational resolution and to program the control mechanisms. This language needed to be efficient enough, since we wanted to get an implementation with acceptable performances. We chose Prolog as implementation language, essentially because it includes a very efficient resolution procedure.

We use an efficient simulation method which avoid to introduce explicitly the axioms of equality. The principle of this method is well-known, and has been used in numerous approaches for introducing functions in logic programming ([Kow83], [Der83], [EY87], [Fri88]).

For this simulation, a logic program is built: to each axiom corresponds a Horn clause without equality. In [Der83] it is shown that, on the clauses obtained by this transformation, the standard depth-first control of Prolog provides an efficient simulation of equational narrowing with leftmost-innermost strategy. This narrowing strategy is comparable to the SLOG resolution strategy. [Fri88] gives a method for transforming a positive conditional specification into a logic program. We use the same transformation.

As in section 2.2, some generators are distinguished among the operations of the specification. A set of generators of s sort is a set of operations $\Omega_s \subset \Sigma$ such that any term of s sort can be proved equal to a term made of generators only. For all s in S, we note $W_\Omega(\mathcal{X})$the term algebra generated from the operations of Ω_s and the variables of $\mathcal{X}$. The operations with results in s which do not belong to Ω_s are called "defined operations." Equational axioms, and conclusions of conditional axioms are implicitly oriented from left to right: the left-hand-side of an equation (or a conclusion of a conditional axiom) is considered as defined by the right-hand-side. Some conditions are necessary to ensure the computational equivalence between resolution and narrowing: the top symbol of the left-hand-side of an equation (or a conclusion) must be a defined operation, and the operands of this operation must belong to $W_\Omega(\mathcal{X})$[Der83]. These conditions forbid, for instance, axioms between generators such as idempotence or commutativity.

We now present the transformation of a specification into a logic program. This transformation works in two steps:

1. The first step yields a set of axioms where any equation (in conclusion or precondition) only contains one occurrence of a defined operation, and it is the top symbol of its left-hand-side.
2. The second step translates these axioms into a set of Horn clauses by replacing each equality by a literal.

4.1 Transforming axioms into Horn clauses

First we give the transformation rules for the first step: they allow to transform positive conditional axioms such that in the transformed conditional axioms all the equations (both in preconditions and conclusions) are of the form:

$$f(t_1, \dots t_n) = t$$

where f is a defined operation and $t, t_1, \dots t_n$ belong to $W_\Omega(\mathcal{X})$.

Then we describe how to obtain Horn clauses from such axioms. Finally, we recall the sufficient conditions on the form of the specification which ensure the equivalence between the equational theory of the initial axioms and SLD resolution on the resulting clauses.

4.1.1 Transformation rules for the first step

Notations: ψ is a conjunction of equations, u is an occurrence in a term, and $t[u \leftarrow t']$ means: t' is the subterm of t at the occurrence u.

r1: Simplification of the right-hand-side of a conclusion

$$\frac{\psi \;\Rightarrow\; l = r[u \leftarrow f(t_1, \dots, t_n)]}{\psi \wedge f(t_1, \dots, t_n) = X \;\Rightarrow\; l = r[u \leftarrow X]}$$

where f is a defined operation and X a new variable.

r2: Elimination of a defined operation at the top of the right hand-side of an equation

$$\frac{\psi \;\wedge\; f(t_1, \dots, t_n) = g(t'_1, \dots, t'_m) \;\Rightarrow\; l = r}{\psi \;\wedge\; f(t_1, \dots, t_n) = X \;\wedge\; g(t'_1, \dots, t'_m) = X \;\Rightarrow\; l = r}$$

where f, g are some defined operations and X is a new variable.

r3: Elimination of an internal occurrence of defined operation

$$\frac{\psi \;\wedge\; t[u \leftarrow f(t_1, \dots, t_n)] = t' \;\Rightarrow l = r}{\psi \wedge f(t_1, \dots, t_n) = X \wedge t[u \leftarrow X] = t' \Rightarrow l = r}$$

where f is a defined operation, u is a strict occurrence in t (i.e. f is not the top symbol of t) and X is a new variable. This rule works both on the right hand-side and on the left-hand-side of an equation.

r4: Elimination of equation between terms of $W_\Omega(\mathcal{X})$

$$\frac{\psi \,\wedge\, t = t' \,\Rightarrow\, l = r}{\sigma(\psi \,\Rightarrow\, l = r)}$$

where t and t' belong to $W_\Omega(\mathcal{X})$and σ is the most general unifier of t and t' (in the free Σ-algebra).

4.1.2 Correctness of the transformation

Rules r1, r2, r3 are just specializations of the following straightforward equivalence:

$$(\forall Y)(\neg(Y = t) \vee \phi) \iff \sigma(\phi)$$

where $=$ is a congruence, Y is a variable not occurring in t, ϕ is any formula and σ is the substitution $\{Y \leftarrow t\}$.

For rule r4, the equivalence between the initial and resulting axioms is only true when the initial equational theory contains no equation between different generators. One sufficient condition ensuring the non-existence of equation between generators in the initial theory is that: there is no axiom defining a generator; and, when orienting from left to right the conclusions of the axioms, the set of axioms is a convergent (confluent and terminating) term rewrite system.

Now let us come back to the rule r4. Under this later condition, if there is no unifier of t and t', then the axiom is meaningless and can be discarded (in our system, a warning is issued to the user in this case).

We have the following properties: rules r2, r3, r4 neither remove nor create cases where rule r1 is applicable; rules r3 and r4 neither remove nor create cases where rule r2 is applicable; rule r4 neither removes nor creates cases where rule r3 is applicable. Thus we choose the following control:

1. apply r1 as long as possible,
2. apply r2 as long as possible,
3. apply r3 as long as possible,
4. apply r4 as long as possible,

It is obvious that, with this control, the transformation terminates. We give in appendix the proof that, up to the symmetry of the equalities occurring in the precondition, the transformed axioms are of the form:

$$f_1(t_{1,1}, \ldots, t_{1,n_1}) = r_1 \wedge f_m(t_{m,1}, \ldots, t_{m,n_m}) = r_m \Longrightarrow f(t_1, \ldots, t_n) = r$$

where $f,\ f_i$ are defined operations and $t_{i,j},\ r_k$ and r belong to $W_\Omega(\mathcal{X})$.

4.1.3 Last step of the transformation

From the axioms above, a set of Horn clauses is built by replacing, in each axiom, every equation $f(t_1, \ldots, t_n) = t$ by a literal $\tilde{f}(t_1, \ldots, t_n, t)$. It means that, with each defined operation f of arity n, is associated a relation name $\tilde{f}$ of arity $n+1$. The last operand of $\tilde{f}$ corresponds to the result of f. The generators are left unchanged.

4.1.4 Examples

Let us consider the classical axioms for defining the addition operation in natural numbers, generated by zero and successor:

add(0, N) = N	-1st step $\rightarrow$	add(0, N) = N
add(s(N), M) = s(add(N, M))		add(N, M) = K $\Rightarrow$ add(s(N), M) = s(K)

where *N, M* and *K* are variables of *Nat* sort. When applying the second step of the transformation, the Horn clauses obtained for these axioms are:

$$add(0, N, N).$$
$$add(s(N), M, s(K)) \text{ :- } add(N, M, K).$$

The binary operation *add* becomes a ternary relation (by notation abuse we still note *add* instead of $\widetilde{add}$). We use the usual clausal notation: $\Leftarrow$ is noted ": $-$" and $\wedge$ is noted ",". The two clauses above are a Prolog program which defines the addition of natural numbers.

We are not only interested in the translation of axioms into Prolog. As seen in section 3, the test data selection implies the resolution of equational problems such as:

$$t_1 = u_1 \wedge \ldots \wedge t_n = u_n$$

Such a problem is transformed into a Prolog goal (negative clause) just as preconditions of conditional axioms in the transformation above.
For instance, the problem

$$add(add(0,\ Y), s(X)) = s(s(Z)) \wedge\ Y = s(T)$$
$$\downarrow$$
$$1^{st} \text{ step}$$
$$\downarrow$$
$$add(0,\ s(T)) = U \wedge add(U,\ s(X)) = s(s(Z)) \text{ with } \sigma : \{Y \leftarrow s(T)\}$$

where *X, Y, Z, T, U* are variables of *Nat* sort. The second step of the transformation leads to the following goal:

$:- \ add(0,\ s(T),\ U),\ \ add(U,\ s(X),\ s(s(Z))).$

with the substitution $\sigma : \{Y \leftarrow s(T)\}$.

4.1.5 SLD resolution versus equational reasoning

We give here some sufficient conditions on the initial specification which ensure the completeness of SLD resolution on the resulting Horn clauses with respect to the equational theory underlying the initial axioms.

These conditions were mentioned initially in [Der83]. They correspond to the ones given in [Fri85] for the completeness of SLOG:

1. When orienting the axiom conclusions from left to right, the resulting conditional rewrite system is terminating (for any term, after a finite number of applications of the rewrite rules, one obtains a normal form, i.e. a term where it is impossible to apply one of the rules) and confluent (for any term, the normal form is unique).
2. There is no axiom *defining* a generator : the top symbol of the left-hand-side of an equation (or a conclusion of a conditional axiom) must be a defined operation.
3. In the left-hand-side of an equation (or a conclusion of conditional axiom), the operands are terms of $W_\Omega(\mathcal{X})$.
4. The specification is complete with respect to the generators: any ground term of W_Σ is equal to a term of W_Ω (which is unique, from the first condition).

When the specification fulfills these conditions, the SLD resolution on the resulting clauses provides a unification procedure correct and complete (for the solutions in normal form) for the equality defined by the initial axioms.

The system we have developed is based on the transformation we have presented, Prolog resolution and some specific control. Thus the specifications it can process must satisfy the four conditions above. Moreover, all the variable instances in the resulting test data sets are in normal form, i.e. belong to W_Ω. This is not restrictive since, as said in section 2, the operations in Ω generate all the values.

4.2 Using a complete search strategy

The standard control strategy in Prolog is depth-first search. It is not satisfactory for our purpose since, when there is an infinite path in the resolution tree of a problem, the solutions which are reachable by some other path may never be generated. As we want to get data sets which are valid with respect to the selection hypotheses discussed in section 3, we need a control strategy which is complete and moreover sufficiently efficient.

A well-known strategy which is complete for our purpose, is the breadth-first search. However, its implementation requires numerous copies of the problem, and complex mechanisms for aliasing of terms, variables, literal, etc.
Thus we have chosen a strategy which is a good compromise between the depth-first search and the breadth-first search: the "iterative depth-first" search consists in stating a bound k for the depth in the resolution tree. When a resolution path reaches this bound, the state of the resolution is stored and the search backtracks to try another choice of clause. Thus the resolution is complete for the solutions which are reachable by a depth less than k in the resolution. If no solution is reached for this bound, if there exist some memorized states of resolution, the process is started again from these states with a new bound $k + k'$ (one can notice that if $k = k' = 1$, it is equivalent to breadth-first search, if $k = \infty$ it is equivalent to depth-first search). As the choice of k and k' strongly depends on the problem to be solved, we have left them as parameters of our system.

4.3 The termination problem

It is well known that Prolog programs do not always terminate, and the choice of the iterative depth-first control is far from solving this problem. For instance, even if this strategy is complete, it does not detect the unsatisfiabilities which correspond to contradictions with the axioms.

Example 25 : Given the logic program defining the addition in natural numbers, let us consider the equational problem:

add(add(s(X), Y), Z) = 0

This is an unsatisfiable problem: the sum of a natural number greater than 0 with an other natural number cannot be equal to 0. The transformation of this problem in a Prolog goal returns:

:− *add(s(X), Y, T), add(T, Z, 0).*

The iterative depth-first search does not terminate on this goal: the resolution of the first literal yields successively for T all the natural numbers, and each time the resolution of the second literal fails.
However, if the initial problem is rewritten via the axioms which define *add*, it becomes:

s(add(add(X, Y), Z)) = 0

In this form, the unsatisfiability is detected at once since there is no equation between different generators in the theory.

From this example, it appears that rewriting should be used as a simplification tool before each resolution step. This simplification not only provides a way to detect some unsatisfiabilities, but it turns out to decrease in a significant way the size of the resolution tree.

Moreover, as the axioms determine a confluent and terminating rewrite system (cf section 4.1) this simplification is deterministic, and it preserves the correctness and com-

pleteness properties. Thus we have integrated this simplification in our system, after adapting it to rewriting of literals, as it is done in [Fri88]. This mechanism was also used in the conditional narrowing algorithms of Rap and Slog.

In the system, the command "rewrite(true)" causes a simplification by rewriting before each resolution step. If the user prefers not to use it, he can use the "rewrite(false)" command.

4.4 Random choice of clauses

As seen in section 3, a uniformity subdomain is defined by a conjunction of equations. Thus, selecting some test data under uniformity hypotheses in a given subdomain requires the resolution of a conjunction of equations. Under uniformity hypotheses, we need only one solution. But it is clear that always retaining the first solution given by the iterative depth-first control is much too deterministic for our purpose: we want an arbitrary value of the subdomain, and very often, the first solution is far from being an arbitrary value. For instance, if an operation is recursively defined with the initial cases first, then the first returned solutions are the values corresponding to those cases. However, we do not require actual randomness in the order of the solution : any non determinism in the order of solutions is quite sufficient.

In order to ensure such a non deterministic order of the solutions, we have implemented a random choice strategy among the clauses which are applicable for a given literal.

Example 26 : Given the operation $< : Nat \times Nat \rightarrow Boolean$ (less than), and the logic program (obtained by transformation of its axioms):

$<(X, 0, false).$
$<(0, s(X), true).$
$<(s(X), s(Y), B)$:–
    $<(X, Y, B).$

We want to solve the equation $<(X, 3) = true$. With depth-first search we get the solutions in the following order:

$X = 0;\ X = 1;\ X = 2;$

with a random-choice strategy for the choice of the clauses, we can get one of the following sequence:

$X = 1;\ X = 2;\ X = 0;$
$X = 2;\ X = 1;\ X = 0;$
$X = 0;\ X = 1;\ X = 2;$

However, all the permutations of these three solutions are not possible, since the back-tracking induces a partial order on solutions.

This is the way we have implemented selection in uniformity subdomain. In our system, this kind of control is activated and disactivated by the commands "random_choice(true)" and "random_choice(false)."

4.5 Decomposition into uniformity subdomains

It remains to provide a tool for the decomposition of the domain of an equational problem into uniformity subdomains. This tool uses some control specifications in order to stop the decomposition. As seen in Section 3, one simple way to do this decomposition is to recursively replace each defined operation occurring in the equational problem (to be decomposed) by the cases corresponding to the axioms which define it.

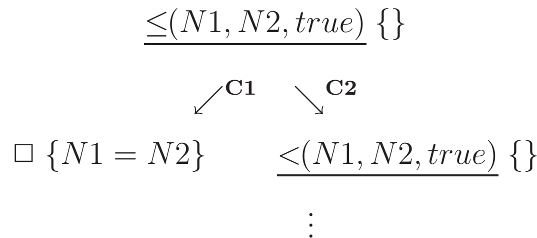
The coverage of the cases occurring in the axioms is already provided by the resolution since all the axioms are tried. Thus, what we need is a mechanism which stops the resolution of a literal when this literal defines a domain where it seems sensible to apply a uniformity hypothesis.

Example 27 : We now consider the operation $\leq$ *: Nat* $\times$ *Nat* $\rightarrow$ *Boolean*, with the corresponding logic program (cf. Section 3):

C1: $\leq(X, X, true)$.
C2: $\leq(X, Y, true)$:– $<(X, Y, true)$.
C3: $\leq(X, Y, false)$:– $<(Y, X, true)$.

Assume we want to decompose the problem $\leq(N1, N2) = true$ until we get equations of the forms $N1 = N2$, $<(N1, N2) = true$.

The resolution of this problem builds the following search tree:



Here, each node of the tree contains a resolvent and a substitution. The edges are labeled by the clause used to solve the underlined literal in the resolvent of the origin node. □ is the empty resolvent, and {} is the identity substitution.

When continuing the construction of the tree under $<(N1, N2, true)$, one will get in the leaves of the tree, the possible values for $N1$ and $N2$.
On this very simple example, it is easy to see that if the resolution of a literal of the form $<(N1, N2, true)$ is stopped, the leaves of the tree contain the decompositions we want (i.e. $N1 = N2$ and $<(N1, N2) = true$.)

Several logic programming languages (Mu-Prolog [Nai82], Metalog [DL84]) allow the programmer to specify the conditions for performing the resolution of a literal. This

induces a strategy for the choice of the literal to be solved which preserves correctness and completeness. This specification is written via meta-clauses which define specific meta-predicates. The meta-clause below defines, via the *wait* meta-predicate, the conditions for delaying the resolution of its argument:

$$wait(<(A, B, true)) \ :\!-\ var(A), var(B).$$

This means that, if the resolvent contains a literal of the form $<(A, B, true)$ and if its operands A and B are variables, then this literal won't be selected for resolution.

We have implemented this control for the choice of the literal to be solved. Thus, the answers of the resolution procedure are couples *(substitution, constraint).* A constraint is a list of literals waiting for subsequent resolution. Moreover, we have introduced a new mode which allows one to force the resolution of constraints. This mode is activated by the command *constraints(false).* The command *constraints(true)* forbids the resolution of the constraints: when a resolvent contains waiting literals only, the resolution is stopped (for this path of the search tree, of course).
When this command is activated, the system builds (from the specification and the wait clauses) a set of uniformity subdomains for each axiom. These uniformity subdomains are defined by a conjunction of constraints and substitutions (where the substitutions are seen as conjunction of equations).

For efficiency reasons, we have introduced a heuristic for the choice of the literals. It gives priority to literals which do not unify with any head of clause, and to those which are unifiable to a unique head of clause.

Summing-up, the selection strategy of the literal to be solved is:

1. choose the first (in sequence) literal of the resolvent which is not unifiable with a head of clause (thus there will be a local failure),
2. then, among the literals of the resolvent which are not affected by a "wait", choose the first literal which is unifiable to a minimum of heads of clauses (deterministic literals have priority)

4.6 Strategies for regularity and uniformity hypotheses

In this subsection, we show how to use this set of tools for implementing some selection strategies compatible with the hypothesis of regularity on the generators of a sort s (Ω_s-regularity) and the uniformity hypothesis.

As in Section 2, we need a canonical complexity measure on terms of s sort in W_Σ that do not contain defined operations of s sort (i.e. terms of s sort belonging to $W_{\Delta\Omega+\Sigma_1+\ldots\Sigma_n}$). We note $complexity_s$ this measure. Its value is the number of occurrences of a generator of s:

$$\forall f : s_1 \times \ldots \times s_n \to s \in \Omega_s,\ complexity_s(f(t_1, \ldots, t_n)) = 1 + \Sigma_{1 \leq i \leq n} complexity_s(t_i)$$

It is not difficult to generate automatically the axioms defining this measure as soon as the signature is given.

Selecting a test data set corresponding to Ω_s-regularity of level n for a variable X, is reduced to the resolution of the equational problem:

$$\leq(complexity_s(X), n) = true$$

which, by transformation becomes:

$$:- \ complexity_s(X, C), \ \leq(C, n, true).$$

It is also possible to use the resolution procedure to select test data sets corresponding to uniformity on a sort *s:* given the generators of *s*, it is possible to construct the axioms defining a typing function: *is_ a_ s : s* $\rightarrow$ *Boolean*

$$\forall f : s_1 \times \ldots \times s_n \rightarrow s \in \Omega_s$$

$$is_a_s_1(t_1) = true \ \wedge \ldots \wedge \ is_a_s_n(t_n) = true \ \Rightarrow \ is_a_s(f(t_1, \ldots, t_n)) = true$$

The application of a uniformity hypothesis on s for a variable X is done by computing the first solution of the goal:

:− *is_ a_ s(X, true).*

with a random strategy for the choice of the clauses.

Example 28 : The generators of the *NatList* sort are *empty* $:\rightarrow NatList$ and *cons* $: Nat \times NatList \rightarrow NatList$. One builds automatically the following definition of *is_ a_ NatList*:

$$is_a_NatList(empty) = true$$
$$is_a_Nat(X) = true \ \wedge \ is_a_NatList(L) = true$$
$$\Rightarrow \ is_a_NatList(cons(X, L)) = true$$

Applying a uniformity hypothesis on *NatList* for a variable L consists in solving (after transformation of the axioms) the goal:

$$:- \ is_a_NatList(L, true).$$

The system will return, for instance:

$L = cons(5, cons(0, empty))$

In our system, the command *make_ tests_ tools("spec_ name")* causes the automatic construction of a complexity function and of a typing function for each sort defined in the specification module *"spec_ name."* The resulting axioms are stored in a specification module *spec_ name_ test* which uses the modules defining the required sorts and operations. For instance, the *NATLIST* specification module defines the *NatList* sort and uses the *Nat* sort. Thus, the module *NATLIST_ test* uses the modules: *NATLIST, NAT_ test, NAT, BOOLEAN.* These two last modules are needed for the complexity function which is of range *Nat* , and for the typing function which is of range *Boolean.*

The next (and last) section of this paper shows how to use our system for selecting test data sets for axioms of the *NATLIST* specification. We have chosen these axiom in order to exercise all the selection strategies presented here.

5 An example of test data set selection using our system

We briefly present the functionalities of our system. Then we show on an example how to use it for selecting test data sets.

5.1 Overview of the system

Our system accepts as input structured positive conditional specifications as defined in Section 2.1. The transformation of the axioms of a specification module into Horn clauses is automatically done when the user asks for compilation of this module. Note that the user never sees the internal form and he/she always works with his/her original specification. In addition, the user can control the resolution strategy defining "wait" expressions for some operations of the module *spec_name* in a module called *spec_name.ctrl* which is automatically consulted when the module *spec_name* is loaded.

The main tool of the system is an interpreter. This interpreter can work in two modes: the command mode, and the request mode.

- In the command mode, it is possible to compile specification modules, to get the "listing" of a specification or of an operation, to generate a specification module including the complexity and typing functions seen in section 4, to modify the various parameters of the resolution procedure via the commands described in the previous section.
- In the request mode, it is possible to solve equational problems of the form:

  :– $eq_1, \ldots, eq_n$.

  where the eq_i are equations between terms of $W_\Sigma(\mathcal{X})$ of the same sort. The solutions obtained are couples of substitution and constraint. The constraint is empty if the command *constraints(false)* has been used: there is no stop on constraints of the resolution.

The transformation of a problem into a conjunction of literals is done automatically when parsing the text of the problem.

The user can ask for one solution only with the strategy of random choice of the clauses, and without stop on constraints (i.e. the selection strategy on a uniformity subdomain) by putting a question mark as a prefix of the problem. For instance:

$:- \ ?(eq_1, \ldots, eq_n\ ).$

This construction can be used anywhere and several times in the same request:

$:- \ pb_1, ?(pb_2), pb_3.$

First, the system computes a couple $(\sigma_1, constraint_1)$ by parameterized resolution of pb_1. Second, it considers the problem $pb_2' = \{constraint_1, \sigma_1(pb_2)\}$, and computes one solution $(\sigma_2, \emptyset)$ with the strategy of random choice of the clauses, and without stop on constraints ($constraint_1$ is solved with $\sigma_1(pb_2)$, giving no constraints as there is no stop on constraints). Finally, it returns a couple $(\sigma_3, constraint_3)$ which is a solution of $pb3' = sigma_2(sigma_1(pb_3))$.

Before starting some test data set selection for the axioms of the *spec* module, the user ask for the creation of a module called *spec_ test* which contains the definitions of the complexity and typing functions for the sorts which are defined in *spec*. He also can modify the module *spec_ test.ctrl* by adding some “wait” meta-clauses.

5.2 Examples of test data set selections

We recall the defining axioms of *sorted* from the specification *NATLIST*.

$$sorted(empty) = true$$
$$sorted(cons(N1, empty)) = true$$
$$sorted(cons(N1, cons(N2, L))) = and(\ \leq(N1, N2), sorted(cons(N2, L)))$$

where $N1, N2$ belong to *Nat* and L belongs to *NatList*. The *and* and $\leq$ operations are defined as in section 3.

We now describe precisely the selection of a test data set for every defining axioms of *sorted*. As the elementary tests are equations between *Boolean* terms, there is no problem of observability: this sort is trivially observable. In non observable cases, the construction of observable contexts for the selected elementary tests can be done after this first selection phase.

We assume that the *NATLIST_ test* specification has been created and loaded (*NATLIST_ test.ctrl* is automatically loaded). All the required specification modules, all the complexity and typing functions are available.

5.2.1 First defining axiom of sorted

There is no variable occurring in this axiom. Thus, the only selected test is:

$$sorted(empty) = true$$

5.2.2 Second defining axiom of sorted

Only one variable occurs in this axiom. It has the imported sort *Nat*. We apply uniformity hypothesis on *Nat*.

For that, we use the typing function $is_a_Nat : Nat \rightarrow Boolean$ which was automatically generated in the *NAT_test* module (c.f. Section 4.6).
We request one solution of the problem *is_a_Nat(N1)* = *true* with the strategy of random choice of the clauses:

$$:- \ ?(is_a_Nat(N1{:}Nat) = true).$$

We get as answer: $N1 = 5$.

Thus the only test for the second axiom of sorted is:

$$sorted(cons(5, empty)) = true$$

5.2.3 Third defining axiom of sorted

We now consider the last defining axiom of *sorted*.

$$sorted(cons(N1, cons(N2, L))) = and(\ \leq(N1, N2), sorted(cons(N2, L)))$$

The $N1$ and $N2$ variables are of the *Nat* sort, which is imported. The L variable is of defined sort *NatList*. Thus we apply a regularity hypothesis (of level 2) to L. For that, we use the complexity function on *NatList* in the following request:

$$:- \ \leq(complexity_{NatList}(L{:}NatList), 2) = true.$$

which returns two solutions:

$$L = empty;$$
$$L = cons(X{:}Nat, empty);$$

We now have two instances of the axioms, where there is no more variable of *NatList* sort:

$$sorted(cons(N1, cons(N2, empty)))$$
$$= and(\ \leq(N1, N2), sorted(cons(N2, empty)))$$
$$sorted(cons(N1, cons(N2, cons(X, empty))))$$
$$= and(\ \leq(N1, N2), sorted(cons(N2, cons(X, empty))))$$

The defining axioms of *and*, $\leq$ and *sorted* give the cases which must be covered. Thus we decompose the right hand sides of the two above instances via these axioms (as in Section 3).

The $\leq$ operation is defined in term of the $<$ operation. Let us assume that we want to cover the corresponding cases and that we want to stop the decomposition at $<$ (i.e. we do not want to analyze the cases introduced by the definition of $<$). We get three uniformity subdomains definitions (see section 3): $M = N$, $<(M, N) = true$ and $<(N, M) = true$.

The computation of this decomposition uses the "stop on constraints" mode. We need to define a "wait" meta-clause which will stop the decomposition of $<$.

$$
\begin{array}{l}
wait(<(N{:}Nat, M{:}Nat) = B{:}Boolean) :- \\
\quad var(B) \rightarrow \\
\quad\quad and(or(var(N), var(M)), N =/= 0, M =/= 0) \\
\quad | \quad B == true \rightarrow \\
\quad\quad\quad var(M) \\
\quad\quad | \quad var(N).
\end{array}
$$

The meaning of $Cond1 \rightarrow Cond2 \mid Cond3$ is **if** *Cond1 is provable* **then** *prove Cond2* **else** *prove Cond3,* and $==$ and $=/=$ are respectively syntactic equality and syntactic difference.

The control of the resolution defined by this meta-clause is: if an equation of the form $<(N, M) = B$ occurs in a resolvent and if N, M, B are not instancied enough for ensuring that this equation has a finite number of solutions, then the resolution of this equation is delayed.

From the form of the defining axioms of $\leq$, it it clear that the resolution of an equation $\leq(M, N) = B$ will lead to equations $<(N, M) = true$, $<(M, N) = true$ or $M = N$ (but $M = N$ appears only on the substitution part of the solution). The above control will block the resolution of the two remaining equations if M and N are not instancied. It is possible to give a simpler control. The interest of the one given above is that it avoids efficiently some cases of non termination; thus it makes the resolution more efficient.

Generally, the statement of the "wait" meta-clauses is not easy. These meta-clauses are not only useful to describe the selection strategy, they also allow one to define efficient control for resolution. Some work on the automatic definition of meta-clauses, in simpler cases, is reported in [Nai85].

Given the meta-clause above, the decomposition of the first instance of the axiom is obtained, using the resolution with stop on constraints, by the request below:

$$:-\ and(\leq(N1{:}Nat, N2{:}Nat), sorted(cons(N2, empty))) = B{:}Boolean.$$

which returns as solutions:

$N1 = N2, B = true;$

$B = true,$
$constraint = \{<(N1, N2) = true\};$

$B = false,$
$constraint = \{<(N2, N1) = true\};$

These solutions correspond to the definitions of uniformity subdomains. To get directly arbitrary solutions in these subdomains, it is possible to use the request:

$$:- \; \mathit{and}(\leq(N1{:}Nat, N2{:}Nat), \mathit{sorted}(\mathit{cons}(N2, \mathit{empty}))) = B{:}Boolean, ?().$$

The ?() construct must immediately follow some equations. After the resolution of these equations, which yields some constraints and substitutions, it activates the selection strategy corresponding to uniformity hypotheses on the subdomains. Thus we get as solutions:

$$N1 = 2, N2 = 2, B = true;$$
$$N1 = 0, N2 = s(X{:}Nat), B = true;$$
$$N1 = s(s(s(s(X{:}Nat)))), N2 = 3, B = false;$$

The two last solutions contain a variable of *Nat* sort. As it is an imported sort, we apply a uniformity hypothesis on *Nat* via the request $?(is_a_Nat(N1) = true, is_a_Nat(N2) = true)$, which is added in the previous request:

$$:- \; \mathit{and}(\leq(N1{:}Nat, N2{:}Nat), \mathit{sorted}(\mathit{cons}(N2, \mathit{empty}))) = B{:}Boolean, ?(),$$
$$?(is_a_Nat(N1) = true, is_a_Nat(N2) = true).$$

The ?() part of the request could be omitted. However, for a better efficiency of the resolution, it is necessary to dissociate the selection in the uniformity subdomains built by decomposition, from the uniformity hypothesis on *Nat*. The final ground solutions are:

$$N1 = 5, N2 = 5, B = true;$$
$$N1 = 1, N2 = 3, B = true;$$
$$N1 = 9, N2 = 7, B = false;$$

Thus we have built the following test data set for the first instance of the axiom (the instance corresponding to $complexity_{NatList}(L) = 1$)

$$\mathit{sorted}(\mathit{cons}(5, \mathit{cons}(5, \mathit{empty}))) = \mathit{and}(\leq(5, 5), \mathit{sorted}(\mathit{cons}(5, \mathit{empty})))$$
$$\mathit{sorted}(\mathit{cons}(1, \mathit{cons}(3, \mathit{empty}))) = \mathit{and}(\leq(1, 3), \mathit{sorted}(\mathit{cons}(3, \mathit{empty})))$$
$$\mathit{sorted}(\mathit{cons}(9, \mathit{cons}(7, \mathit{empty}))) = \mathit{and}(\leq(9, 7), \mathit{sorted}(\mathit{cons}(7, \mathit{empty})))$$

For the second instance, which corresponds to $complexity_{NatList}(L) = 2$, the method is similar. The request below realize a selection strategy compatible with the hypotheses used for the first instance:

$$:- \; \mathit{and}(\leq(N1{:}Nat, N2{:}Nat), \mathit{sorted}(\mathit{cons}(N2, \mathit{cons}(X{:}Nat, \mathit{empty})))) = B{:}Boolean,$$
$$?(), ?(is_a_Nat(N1) = true, is_a_Nat(N2) = true, is_a_Nat(X) = true).$$

The resulting test data set for the second instance is:

$$sorted(cons(2, cons(2, cons(2, empty)))) = \\ and(\leq(2,2), sorted(cons(2, cons(2, empty))))$$
$$sorted(cons(0, cons(0, cons(4, empty)))) = \\ and(\leq(0,0), sorted(cons(0, cons(4, empty))))$$
$$sorted(cons(4, cons(4, cons(2, empty)))) = \\ and(\leq(4,4), sorted(cons(4, cons(2, empty))))$$
$$sorted(cons(0, cons(1, cons(1, empty)))) = \\ and(\leq(0,1), sorted(cons(1, cons(1, empty))))$$
$$sorted(cons(0, cons(2, cons(3, empty)))) = \\ and(\leq(0,2), sorted(cons(2, cons(3, empty))))$$
$$sorted(cons(1, cons(2, cons(0, empty)))) = \\ and(\leq(1,2), sorted(cons(2, cons(0, empty))))$$
$$sorted(cons(3, cons(0, cons(0, empty)))) = \\ and(\leq(3,0), sorted(cons(0, cons(0, empty))))$$
$$sorted(cons(5, cons(0, cons(5, empty)))) = \\ and(\leq(5,0), sorted(cons(0, cons(5, empty))))$$
$$sorted(cons(6, cons(2, cons(1, empty)))) = \\ and(\leq(6,2), sorted(cons(2, cons(1, empty))))$$

The test data set for the third defining axiom of sorted is the union of the two data sets above.
By construction, the corresponding testing context refinement is the following: the regularity and uniformity hypotheses mentioned above are added to the hypotheses; in the test data set, the third defining axiom is replaced by the selected elementary tests above; and the oracle is the built-in equality on booleans.

When all the axioms of the *NATLIST* specification module are treated in the same way, the resulting testing context is practicable, as proved in Section 2.

One can remark that it is possible to "program" complex selection strategies with one request only. For instance, in our example, there is no necessity to separate the selection step which corresponds to regularity, as we have done for explanatory purposes. The global selection strategy, i.e. regularity of level 2, uniformity on the subdomains obtained by decomposition, uniformity on the remaining *Nat* variables can be realized by the following request:

$$:- \leq(complexity_{NatList}(L{:}NatList), 2) = true, \\ and(\leq(N1{:}Nat, N2{:}Nat), sorted(cons(N2, L))) = B{:}Boolean, \\ ?(), ?(is_a_Nat(N1) = true, is_a_Nat(N2) = true, is_a_NatList(L) = true).$$

The resolution of the equation $is_a_NatList(L) = true$ makes it possible to apply a selection strategy corresponding to a uniformity hypothesis on all the subterms of *Nat*

sort occurring in L. The solutions of this request allow to build a test data set similar to the previous one. Of course the instances selected by uniformity may differ.

The system we have presented here provides a sort of kernel language for programming test data selection from algebraic specifications. This kernel has been carefully designed, on sound theoretical bases: we think it is a good starting point for the development of an integrated, well interfaced environment for designing test data sets from formal specifications (as a by-product, it would be a very nice prototyping environment).

The example we have presented here is not a toy example: generating relevant test data sets for sorted lists is difficult, more difficult than most of the practical problems. At this moment, the system is experienced on a large specification, namely the embedded software of the automatic pilot system of the Lyon subway [DM91], and some data sets have been generated for some modules of the specification. It is too early to state conclusions, but until now, only simple requests have been necessary for this industrial example.

Comparison with previous and related works

The work reported here is the continuation of the works reported in [BCFG85], [BCFG86], [Cho86], [GM88], [Mar89] on the generation of test data sets from algebraic specifications using logic programming.

Even if there is some general agreement that using logic programming for assisting program testing is a good idea [Ger85][PSSS85], and that deriving test data sets from specification is interesting [Rig85][Scu88][Rub89] , there are few other published works in this area.

A pioneering work on test methods based on algebraic specifications was [GMH81] where the idea of exercising the axioms, via the procedures of the program under test, was first presented.

More recently, [GCG85] used the uniformity and regularity hypotheses, as defined in [BCFG85] for testing a Unix-like directory structure against a OBJ-UMIST specification.

In [Wil88], the idea of using constraints, first advocated in [Cho86], is pushed further; Generic Constraint Logic Programming is experienced with an aim different from ours, which is to address incompleteness in analysis.

The main advances reported in this paper are:

- the definition of the concept of a testing context as a triple (hypotheses, test data set, oracle), and of a refinement preorder on such testing contexts;
- the introduction of oracle hypotheses, and more generally of a satisfactory way of coping with the oracle problem (it was not the case in [BCFG86]);
- an attempt to enlarge the class of formal specifications considered. As shown in section 1, this generalization works well but for the unbias properties and the oracle

problem which are dealt with in the specialized framework of algebraic specifications. We intend to continue in this direction;

- the replacement of the notion of generation by the notion of selection: it allows a very nice justification of the way the test data sets are refined;
- the existence of a specialized system which allows to program various selection strategies, which are justified theoretically.

Acknowledgements
This work has been partially supported by the Meteor Esprit Project and the PRC "Programmation et Outils pour l'Intelligence Artificielle." It is a pleasure to thank Laurent Fribourg for numerous helpful discussions.

Appendix: Form of the axioms resulting from the first step of the transformation

We have to prove that the axioms resulting from the transformation presented in section 4.1 are of the form:

$$f_1(t_{1,1}, \dots, t_{1,n_1}) = r_1 \wedge f_m(t_{m,1}, \dots, t_{m,n_m}) = r_m \Longrightarrow f(t_1, \dots, t_n) = r$$

where f, f_i are defined operations and $t_{i,j}$, r_k and r belong to $W_\Omega(\mathcal{X})$.

- The form of the left hand-side of the conclusion is a direct consequence of the form of the initial axiom.
- For the right hand-side of the conclusion, r, if it contains an occurrence of a defined operation, r1 is applicable; this is impossible since the control apply r1 while possible.
- For the equalities of the precondition: let us note $s_1 = s_2$ such an equality.
  1. If both s_1 and s_2 are $W_\Omega(\mathcal{X})$ terms, r4 is applicable.
  2. Else, if either s_1 or s_2 contains a strict occurrence of a defined operation, r3 is applicable.
  3. Else, if both the top symbols of s_1 and s_2 are defined operations, r2 is applicable.

  In these three cases there is a contradiction with the control. Thus either s_1 or s_2 belong to $W_\Omega(\mathcal{X})$. Then, up to the symmetry of these equalities, all the equalities in the precondition are of the defined form.

References

[ADJ76] Goguen J., Thatcher J., Wagner E. : "*An initial algebra approach to the specification, correctness, and implementation of abstract data types*" Current Trends in Programming Methodology, Vol.4, Yeh Ed. Prentice Hall, 1978.

[BCFG85] Bougé L., Choquet N., Fribourg L., Gaudel M. C. : "*Application of PROLOG to test sets generation from algebraic specifications*" Proc. International Joint Conference on Theory and Practice of Software Development (TAPSOFT), Berlin (R.F.A), Springer-Verlag LNCS 186, p.246-260, March 1985. Also: Research Report LRI No.176.

[BCFG86] Bougé L., Choquet N., Fribourg L., Gaudel M. C. : "*Test sets generation from algebraic specifications using logic programming*" Journal of Systems and Software Vol.6, No.4, p.343-360, November 1986. Also: Research Report LRI No.240.

[Ber89] Bernot G. : "*A formalism for test with oracle based on algebraic specifications*" LIENS Report 89-4, LIENS/DMI, Ecole Normale Supérieure, Paris, France, May 1989.

[Cho86] Choquet N. : "*Test data generation using a PROLOG with constraints*" Workshop on Software Testing, Banff Canada, IEEE Catalog Number 86TH0144-6, p.132-141, July 1986.

[DM91] Dauchy P., Marre B. : "*Test data selection from the algebraic specification of a module of an automatic subway*" LRI report No.638, LRI, Université Paris-sud, Orsay, France, january 1991.

[Der83] Deransart P. : "*An Operational Algebraic Semantics of PROLOG Programs*" Proc. Programmation en Logique, Perros-Guirrec, CNET-Lannion, March 83.

[DL84] Dincbas M., Le Pape J.P. : "*Metacontrol in logic programs in METALOG*" 5th Generation Conf., Tokyo, November 1984.

[EY87] Van Emden M. H., Yukawa K. : "*Logic Programming with Equations*" J. Logic Programming (4), p.265-268, 1987.

[Fri85] Fribourg L. : "*SLOG, a Logic Programming Language Interpreter Based on Clausal Superposition and Rewriting*" International Symposium on Logic Programming, Boston, July 1985.

[Fri88] Fribourg L. : "*Prolog with Simplification*" Programming of Future Generation Computers. Fuchi, Nivat Ed. Elsevier Science Publishers B.V. (North Holland), 1988.

[Ger85] Gerhart S. : "*Software Engineering Perspectives on Prolog*" Technical report TR-85-13, Wang Institute of Graduate Studies, July 1985.

[GCG85] Gerrard C. P., Coleman D., Gallimore R. : "*Formal Specification and Design Time Testing*" Software Science ltd, technical report, June 1985, IEEE trans. on Software Engineering, vol.16, No.1, january 1990.

[GG75] Goodenough J.B., Gerhart S.L. : "*Towards a theory of test data selection*" IEEE trans. soft. Eng. SE-1, 2, 1975. Also: SIGPLAN Notices 10 (6), 1975.

[GH86] Geser, Hussmann : "*Experiences with the RAP system - a specification interpreter combining term rewriting and resolution techniques*" Proc. ESOP 86 Conf., LNCS 213, p.339-350, 1986.

[GMH81] Gannon J., McMullin P., Hamlet R. : "*Data-Abstraction Implementation, Specification, and Testing*" ACM transactions on Programming Languages and Systems, Vol.3, No.3, p.211-223, July 1981.

[GM88] Gaudel M.-C., Marre B. : "*Algebraic specifications and software testing: theory and application*" LRI Report 407, Orsay, February 1988, and extended abstract in Proc. workshop on Software Testing, Banff, IEEE-ACM, July 1988.

[Hen91] Hennicker R. : "*Observational implementation of algebraic specifications*" Acta Informatica, vol.28, No.3, p.187-230, 1991.

[Hus85] Hussmann : "*Unification in Conditional-Equational Theories*" Technical Report MIP-8502, Univ. Passau, January 1985, and short version in Proc. EUROCAL 85 Conf., Linz.

[Kam83] Kamin S. : "*Final Data Types and Their Specification*" ACM Transactions on Programming Languages and Systems (5), p.97-123, 1983.

[Kap84] Kaplan S. : "*Conditional rewrite rules*" Theoretical Computer Science (33), p.175-193, December 1984.

[Kow83] Kowalski R. : "*Logic Programming*" Proc. IFIP 1983, p.133-145.

[Mar89] Marre B. : "*Génération automatique de jeux de tests, une solution : Spécifications algébriques et Programmation logique*" Proc.Programmation en Logique, Tregastel, CNET-Lannion, p.213-236, May 1989.

[Nai82] Naish L. : "*An introduction to MU-PROLOG*" Technical report, 82/2, Dept. of Computer Science, Univ. of Melbourne, 1982.

[Nai85] Naish L. : "*Negation and Control in Prolog*" LNCS 238, (Springer-Verlag), 1985.

[PSSS85] Pesch H., Schnupp P., Schaller H., Spirk A.P. : "*Test Case Generation using Prolog*" ICSE 8, London, 1985, p.252-258.

[Rig85] Rigal G. : "*Generating Acceptance Tests from SADT/SPECIF*" IGL technical report, August 1986.

[Rub89] Rubaux J.P. : "*Rapport final d'étude du poste de generation de test ASA*" RATP Paris, Direction des Equipements Electriques, TT-SEL/89-183/JPR.

[Scu88] Scullard G. T. : "*Test Case Selection using VDM*" VDM'88, Dublin, 1988, LNCS 328, p.178-186.

[Sch86] Schoett O. : "*Data abstraction and the correctness of modular programming*" Ph. D. Thesis, Univ. of Edinburgh, 1986.

[ST87] Sannella D., Tarlecki A. : "*On observational equivalence and algebraic specification*" Journal of Computer and System Sciences 34, p.150-178, 1987.

[Wey80] Weyuker E. J. : "*The oracle assumption of program testing*" Proc. 13th Hawaii Intl. Conf. Syst. Sciences 1, p.44-49, 1980.

[Wey82] Weyuker E. J. : "*On testing non testable programs*" The Computer Journal 25, 4, p.465-470, 1982.

[Wil88] Wild C : "*The use of Generic Constraint Logic Programming for Software Testing and Analysis*" Comp. Sc. Technical Report Series, No.88-02, Dept of Computer Science, Old Dominion University, Norfolk, VA, February 1988. Extended abstract in Proc. 2nd ACM-IEEE Workshop on Software Testing, Verification, and Analysis, Banff, July 1988.

Chapitre 5 :

Proving the correctness of algebraically specified software: modularity and observability issues

Gilles BERNOT & Michel BIDOIT

LIENS, CNRS URA 1327
Ecole Normale Supérieure,
45 Rue d'Ulm,
F-75230 PARIS Cédex 05,
FRANCE

bitnet: [bernot,bidoit]@frulm63
uucp: [bernot,bidoit]@ens.ens.fr

(Appeared in Proc. of the AMAST Conference, 1991. To appear in LNCS.)

Contents

1 Introduction **142**

2 Modularity and software correctness **143**

3 The stratified loose approach **144**

4 Observability and software correctness **150**

5 Observational specifications **152**

5.1 Definitions 154

5.2 Initiality results 156

5.3 Structured observational specifications 157

6 Modular observational specifications **159**

7 Specifying adequate observations **163**

8 Conclusion **164**

Abstract

We investigate how far modularity and observability issues can contribute to a better understanding of software correctness. We detail the impact of modularity on the semantics of algebraic specifications and we show that, with the stratified loose semantics, software correctness can be established on a module per module basis. We discuss observability issues and we introduce an observational semantics where sort observation is refined by specifying that some operations do not allow observations. Then the stratified loose approach and our observational semantics are integrated together. As a result, we obtain a framework (modular observational specifications) where the definition of software correctness is adequate, i.e. fits with actual software correctness.

1 Introduction

A fundamental aim of formal specifications is to provide a rigorous basis to establish software correctness. Indeed, it is well-known that proving the correctness of some piece of software without a formal reference document makes no sense.[1] Algebraic specifications are widely advocated as being one of the most promising formal specification techniques. However, to be provided with some algebraic specification is not sufficient per se. A precise (and adequate) definition of what does mean the correctness of some piece of software w.r.t. its algebraic specification is mandatory. This crucial prerequisite must be first fulfilled before one can develop relevant verification methods, and try to mechanize them.

Hence the adequacy of the chosen definition of correctness has a great practical impact, and we should therefore define software correctness in conformity with actual needs. In the framework of algebraic specifications, straightforward definitions of correctness turn out to be oversimplified: most programs that must be considered as being correct (from a practical point of view) are rejected. Indeed, when the program behaves correctly, there can still exist some differences between the properties stated by the specification and those verified by the program. Here, to behave correctly means that these differences are not "observable". Consequently, more elaborated definitions of correctness, taking observability into account, should be considered.

As soon as real-sized systems are involved, both the specification and the software become large and complex. Hence the validation process becomes itself an unmanageable task. At the programming level, this problem is handled by using modular programming languages. At the specification level, algebraic specifications are split into smaller units by means of specification-building primitives. Thus, with respect to software correctness, what is needed is a framework such that the various units of the specification can be related to the various modules of the software, and such that the gobal correctness of the software can be established from the local correctness of each software module w.r.t. its specification module.

[1] Who would attempt to prove a theorem without providing its statement?

Thus, our claim is that modularity and observability issues are fundamental to define a practicable notion of software correctness. In this paper, we will detail various aspects related to modularity, observability, and their interactions with software correctness. We introduce a semantic framework for modular observational algebraic specifications that leads to a more adequate definition of software correctness. This is only a first step towards putting software correctness proofs in practice, but we believe that practicable proof methods can be developed on top of our approach.

We assume that the reader is familiar with algebraic specifications [GTW78, EM85] and with the elementary definitions of category theory [McL71]. An algebraic specification *SP* is a tuple $(S, \Sigma, \mathcal{A}x)$ where (S, Σ) is a signature and $\mathcal{A}x$ is a finite set of Σ-formulas. We denote by $Mod(\Sigma)$ the category of all Σ-algebras, and by $Mod(SP)$ the full subcategory of all Σ-algebras for which $\mathcal{A}x$ is satisfied. We will also use the following technical definition:

Definition 1 : (Minimal models)
Given a specification *SP*, a model $M \in Mod(SP)$ is called *minimal* if, for all $A \in Mod(SP)$, if there exists a morphism $\mu : A \longrightarrow M$, then there exists a **unique** morphism $\nu : M \longrightarrow A$.
Note that if $Mod(SP)$ has an initial model, then it is the unique minimal model (up to isomorphism).

2 Modularity and software correctness

In this section we shall focus on the links that can (should) be established between a modular specification and the corresponding software, implemented using a modular programming language (such as e.g. Ada, Clu or Standard ML). The problem considered is to define an algebraic semantic framework such that the various pieces of the specification can be related to the various modules of the implementation and such that the global correctness of the implementation can be established from the local correctness of each software module w.r.t. its specification module.

To better understand why and how far both the modularity of the specification and the modularity of the software interact together as well as the need for a new approach to the semantics of algebraic specifications, we shall first briefly recall the paradigm of the loose approach.

A specification is supposed to describe a future or existing system in such a way that the properties of the system (**what** the system does) are expressed, and the implementation details (**how** it is done) are omitted. Thus a specification language aims at describing **classes** of correct (w.r.t. the intended purposes) implementations (realizations). In contrast a programming language aims at describing **specific** implementations (realizations). In a loose framework, the semantics of some specification *SP* is a class $\mathcal{M}$ of (non-isomorphic) algebras. Given some implementation (program) P, its correctness w.r.t. the specification *SP* can then be established by relating the program P with one of the algebras of the class $\mathcal{M}$. Roughly speaking, the program P will be correct w.r.t.

the specification SP if and only if the algebra defined by P belongs to the class $\mathcal{M}$.[2]

Let us now reexamine the above picture in a modular setting. At one hand we have a **modular** specification SP made of some specification modules ΔSP_1, ΔSP_2, ... related to each other by some specification-building primitives. On the other hand we have a **modular** software made of some program modules ΔP_1, ΔP_2, ... Assume moreover that the software structure reflects the specification structure. The problem we have to solve is the following one:

1. To define a notion of correctness such that "the program module ΔP_i is correct w.r.t. the specification module ΔSP_i" is given a precise meaning.
2. To ensure that the local correctness of each program module w.r.t. its specification module implies the global correctness of the whole software w.r.t. the whole specification.
3. To carefully study how some basic requirements about the modular development of modular software, as well as their reusability, interact with the design of the semantics of modular specifications.

It turns out that the main difficulties raised by this goal are twofold:

1. Providing a (loose) semantics to specification **modules** is not so easy, since from a mathematical point of view (heterogeneous) algebras do not have a modular structure.
2. If our intuition and needs about modular software development and the reuse of software modules can be easily figured out, this turns out to be a more difficult task at the level of algebraic semantics.

In the following section we shall try to provide some insight into the solution we propose and into the main ideas underlying what we call the "*stratified loose semantics*".

3 The stratified loose approach

For sake of simplicity, we shall focus on the most commonly used specification-building primitive, namely the enrichment one. Moreover, we shall assume that the modular specification SP_2 we consider is made of one specification module ΔSP that enrich only one modular specification SP_1.

[2] As we will see in Section 4, this is an oversimplified picture. However, in the sequel we shall adopt this oversimplified understanding of software correctness, since it will be sufficient to study the impact of modularity. Note also that our picture does not preclude more refined views about implementations, such as the abstract implementation of one specification by another (more concrete) one [BBC86, EKMP82], or the stepwise refinement and transformation of a specification into a piece of software [CIP85]. This indeed is the reason why whe shall speak of "realizations" instead of "implementations".

According to the loose approach, the semantics of the specification SP_1 will be defined as some class $\mathcal{M}_1$ of Σ_1-algebras (where Σ_1 denotes the signature associated to SP_1). Similar notations hold for SP_2. Since we assume that SP_2 is defined as an enrichment of SP_1 by the specification module ΔSP, we have $\Sigma_2 = \Sigma_1 \oplus \Delta\Sigma$, hence $\Sigma_1 \subset \Sigma_2$. Let $\mathcal{U}$denote the usual forgetful functor from Σ_2-algebras to Σ_1-algebras.

With the help of this simple context, our intuition and needs w.r.t. the modular development of modular software can be summarized as follows [Bid87]:

1. If some piece of software fulfills (i.e. is a correct realization of) the "large" specification SP_2, then it must be reusable for simpler purposes, i.e. it must also provide a correct realization of the sub-specification SP_1.
2. **Any** piece of software that fulfills (i.e. that is a correct realization of) the sub-specification SP_1 should be reusable as the basis of some correct realization of the larger specification SP_2. In other words, it should be possible to implement the sub-specification SP_1 without taking care of the (future or existing) enrichments of this specification (e.g. by the specification module ΔSP).
3. It should be possible to implement the specification module ΔSP without knowing which specific realization of the sub-specification SP_1 has been (or will be) chosen. Thus, the various specification modules should be implementable **independently** of each other, may be simultaneously by separate programmer teams. Moreover, exchanging some correct realization (say P_1) of the specification SP_1 with another correct one (say P_1') should still produce a correct realization of the whole specification SP_2, without modification of the realization ΔP of the specification module ΔSP.

The first two requirements can be easily achieved by embedding some appropriate *hierarchical constraints* into the semantics of the enrichment specification-building primitive. Roughly speaking, it is sufficient to require the following property:

> *Either $\mathcal{M}_2 = \emptyset$ (in that case the specification module ΔSP will be said to be hierarchically inconsistent) or $\mathcal{U}(\mathcal{M}_2) = \mathcal{M}_1$.*

The third requirement, however, cannot be achieved without providing a suitable (loose) semantics to **specification modules**. There is no way to take this requirement into account by only looking at the semantics of specifications. However, in an initial approach to algebraic semantics (cf. e.g. [EM85, EFH83]), an initial semantics can be provided for the specification module ΔSP by considering the free synthesis functor $\mathcal{F}_\Delta$ (left adjoint to the forgetful functor $\mathcal{U}$). In our case, nothing ensures that this free synthesis functor $\mathcal{F}_\Delta$ exists, since we have made no assumption about the axioms of the specification. Moreover, we are looking for a loose semantics of specification modules, in order to reflect **all** correct implementation choices of these modules. The following definition provides the solution we are looking for by embedding the ideas of the initial approach into the loose one:

Definition 2 : (Stratified loose semantics)
Given a modular specification SP_2 defined as the enrichment of some modular spec-

ification SP_1 by a specification module ΔSP, the semantics of the specification module ΔSP and of the modular specification SP_2 are defined as follows:

Basic case:

If the sub-specification SP_1 is empty (hence the specification SP_2 is reduced to the specification module ΔSP), then:

- The semantics of the specification SP_2 is by definition the class of all minimal models of $Mod(SP_2)$, if any; if $Mod(SP_2)$ has no minimal model, then SP_2 is said to be *inconsistent*.
- The semantics of the specification module ΔSP is defined as being the class of all functors $\mathcal{F}$ from the category $\mathbf{1}$ to $Mod(SP_2)$, which map the object of $\mathbf{1}$ to a minimal model of $Mod(SP_2)$.[3]

General case:

Let us denote by $\mathcal{M}_1$ the class of models associated to the modular specification SP_1, according to the current definition.

- The semantics of the specification module ΔSP is defined as being the class $\mathcal{F}_1^2$ of all the mappings $\mathcal{F}$ such that:
  1. $\mathcal{F}$ is a (**total**) functor from $\mathcal{M}_1$ to $Mod(SP_2)$.
  2. $\mathcal{F}$ is a right inverse of the forgetful functor $\mathcal{U}$, i.e.: $\forall M_1 \in \mathcal{M}_1 : \mathcal{U}(\mathcal{F}(M_1)) = M_1$.

  If the class $\mathcal{F}_1^2$ is empty, then the enrichment is said to be *hierarchically inconsistent*.
- The semantics of the whole specification SP_2 is defined as being the class of all the models image by the functors $\mathcal{F}$ of the models of $\mathcal{M}_1$: $\mathcal{M}_2 = \bigcup_{\mathcal{F} \in \mathcal{F}_1^2} \mathcal{F}(\mathcal{M}_1)$.

The class $\mathcal{M}_2$ of the models of the specification SP_2 is said to be **stratified** by the functors $\mathcal{F}$.

Some comments are necessary to better understand the previous definition:

- Our semantics is a true loose semantics, since it associates a class of (non-isomorphic) functors (resp. algebras) to a given specification module (resp. to a given specification). However, our semantics can also be considered as a generalization of the initial approach: if we restrict to positive conditional equations, then the free synthesis functor from $Mod(SP_1)$ to $Mod(SP_2)$ exists; under suitable additional assumptions, this functor is just one specific functor in the class $\mathcal{F}_1^2$.

- It is important to note that with our loose stratified semantics, the *hierarchical constraints* mentioned above are satisfied. More precisely, as soon as the specification module ΔSP is hierarchically consistent, then we have $\mathcal{U}(\mathcal{M}_2) = \mathcal{M}_1$. As a consequence, both the so-called "*no junk*" and "*no confusion*" properties are guaranteed.

[3] As usual, the category $\mathbf{1}$ denotes the category containing only one object, which can be interpreted as a Σ_1-algebra for an empty signature Σ_1.

In other words, we know that the "old" carrier sets (i.e. the carrier sets of sorts defined in SP_1) will contain no "new" value, and that "old" values who may be distinct before (in at least one model of SP_1) should not be forced to be equal by the new specification module ΔSP.

- We have chosen a pseudo initial semantics (minimal models) for the basic case (a modular specification reduced to one specification module) in order to exclude trivial models (a well-known problematic feature of loose semantics). This remark does not only apply for basic specifications, but for all modular specifications in general, since this "minimal" semantics for the basic case, combined with the hierarchical constraints induced by the stratified loose semantics for the general case, will exclude trivial carrier sets for all sorts.[4] Moreover, such a "minimal" semantics for basic specification modules turns out to be quite adequate to specify enumerated sets (such as e.g. booleans, characters, etc.).
- So far, we have stressed that the independent implementability of each specification module is a crucial aspect of modularity. Now, we would like to stress another equally important aspect of modularity, namely the specification style point of view. Indeed, when writing some specification module, a natural implicit assumption made by the specifier is that the semantics of the imported sub-specifications is preserved (i.e. this semantics is established once for all). By the way, this is exactly what is guaranteed by our stratified loose semantics: as explained aboved, the hierarchical constraints associated to our semantics of modularity automatically restrict the class of models of the specification SP_2 to the models that preserve the enriched sub-specifications. This contrasts with a more conventional approach, where the specifier should explicitly design the axioms of the specification unit in order to guarantee the so-called no junk and no confusion properties, i.e. to guarantee the persistency of the enrichment (and this often results in unnecessary over-specification). From our point of view, there is therefore a fundamental distinction between what we call **structured specifications**, for which the no junk and no confusion properties are **explicitly** ensured by appropriate axioms, and **modular specifications**, for which similar properties are **implicitly** ensured by an appropriate semantics. Furthermore, it is clear that the hierarchical structure of a modular specification has a deep impact on its modular semantics, while this is not the case for (persistent) structured specifications, whose semantics is not altered by flattening.
- The extension of the definition above to the case where the specification module ΔSP enriches more than one specification as well as its extension to other specification-building primitives (such as e.g. parameterization) do not raise difficult problems and is described in [Bid89].
- It is also important to note that our definition is independent of the underlying institution [GB84]. Thus our stratified loose approach can be used to define the semantics of any modular algebraic specification language [ST84]. Moreover, our

[4] More precisely, if we consider a sort s and if we assume that there exists some operation (or some composition of operations) which has s in its domain and a sort s' defined in a basic specification module as its codomain, then the "minimal" semantics of the basic specification module will prevent from undesired confusion of values in the carrier set of s ...

stratified loose approach can even be used in a more general framework than institutions, for instance in a framework where the *Satisfaction Condition* [GB84] does not hold. This is obvious since, as far as the stratified loose semantics is concerned, the existence of forgetful functors (from Σ_2-algebras to Σ_1-algebras, with $\Sigma_1 \subset \Sigma_2$) is the only requirement. Indeed, this very broad scope of the stratified loose approach will be clearly demonstrated in Section 6.

We must now point out how far our stratified loose semantics solves the problem stated in the previous section. A program module ΔP will be said to be correct w.r.t. some specification module ΔSP if and only if ΔP "defines" a functor belonging to the semantics of the specification module. From our definition, it is then obvious that the "composition" of correct software modules (i.e. the software obtained by linking together these software modules) is always a correct realization of the whole specification. Thus, the main significance of the stratified loose framework outlined in this section is that it is possible to specify and develop software in a modular way, and that the correctness of the implementation should only be established on a module per module basis. A formal theory of software reusability, built on top of our stratified loose semantics, is described in [GM88].

Note that the definition above is given in a very general way: we have considered **all** algebras, finitely generated or not. It is obviously very easy to refine our definition of the stratified loose semantics in order to consider finitely generated algebras only. Indeed, we prefer to introduce a more powerful constraint, namely the restriction to algebras finitely generated with respect to a distinguished subset of the signature called the set of *generators*.[5] Such a constraint will guarantee that for any model, all values will be denotable as some composition of these generators. From a theoretical point of view, an important consequence of this constraint is that *structural induction restricted to the generators* is a correct proof principle. This constraint has many practical consequences too, since reasoning by means of generators helps writing the axioms in a structured way [Bid82]. Moreover, it can be used to avoid to overspecify some operations, as demonstrated in the following two examples:

Specifying a *remove* operation on sets :

To specify a *remove* operation on sets, the following axioms are sufficient:

$$x \in remove(x, S) = false$$
$$x \neq y \implies y \in remove(x, S) = y \in S$$

Considering only finitely generated models w.r.t. the generators will ensure that, for any set S, $remove(S)$ is actually a set, reachable from the empty set by some successive insertions, and not a "junk" set.

Specifying the Euclidean division :

Similarly, to specify the division of natural numbers, the following axioms are sufficient:

[5] We do not detail here the refined version of the stratified loose semantics according to this additional constraint, since the modifications to be introduced are rather obvious [Bid89].

$$m \neq 0 \implies 0 \leq [n - ((n\ div\ m) * m)] = true$$
$$m \neq 0 \implies m \leq [n - ((n\ div\ m) * m)] = false$$

These axioms characterize $(n\ div\ m)$ among all natural numbers *finitely generated w.r.t.* 0 *and succ* (Euclid). However, without the constraint, there are models (e.g. the initial model) where $(n\ div\ m)$ is not reached by some $succ^i(0)$; it is only an unreachable value such that the (unreachable) remainder $[n - ((n\ div\ m) * m)]$ returns the specified boolean values when compared with 0 and m .

In Pluss [Bid89], the distinguished subset of generators is specified apart from the other operations of the signature and is introduced by the keyword **generated by**.

A crucial issue is obviously to know when some given specification module is *hierarchically consistent*. From a general point of view, it is well-known that this is an undecidable problem. However, we would like to point out that, in our stratified loose framework, there are two distinct grounds for hierarchical inconsistency:

- As usual, hierarchical inconsistency may result from the axioms introduced by the specification module.
- Moreover, adding **"new" observations on "old" sorts** will in general result in an inconsistent specification module ΔSP. This is due to the fact that, with these "new" observations, it may be possible to distinguish "old" values (i.e. to prevent them from being equal), while these "old" values could have been equal in some model M_1 of SP_1. Hence there is no **total** mapping from $\mathcal{M}_1$ to $Mod(SP_2)$, since this model M_1 of SP_1 cannot be extended to a model of SP_2. A typical example of such a situation is when a "new observing operation" on an "old" sort is defined in ΔSP: for instance, if we assume that SP_1 specifies natural numbers (with $\mathcal{M}_1 \supseteq \{\mathbb{N},\ \mathbb{Z}/n\mathbb{Z}\}$), specifying a "$\leq$" operation in ΔSP will result in an hierarchically inconsistent specification module. Thus, these "observing operations" should rather have been defined in the appropriate specification module, i.e. in the specification module where the "observed" sort is defined.

Note that the latter ground for hierarchical inconsistency should not be understood as a restrictive side-effect of the stratified loose semantics, but rather as a fruitful guide in structuring large specifications into specification modules. More precisely, a specification module should be considered as a unit of specification where a sort of interest, its generators, its observers, and other appropriate operations are simultaneously defined.

As a consequence of the "hierarchical constraints" required by modularity, it is necessary to state a careful distinction between *implementable* and *not yet implementable* specification modules:

- *Implementable specification modules* will have a semantics defined accordingly to the stratified loose framework, in order to allow for a **modular** software development and verification process.
- *Not yet implementable specification modules* will have a more flexible semantics, in order to allow for a specification development process by stepwise refinements [Bid91].

Such a distinction is introduced in the **Pluss** algebraic specification language [Bid89, BGM89], the semantics of which is defined following the stratified loose approach. Note that this distinction contrasts with all other specification languages developed following either the initial or the loose approach, such as ASL [Wir83, AW86], OBJ2 [FGJM85] and LARCH [GHW85], where there is only a distinction between various enrichment primitives.

4 Observability and software correctness

The intuition used in Section 2 to introduce the stratified loose approach was obviously an oversimplified understanding of software correctness. Indeed, if software correctness (w.r.t. its formal specification) is defined in such a way, then most realizations that we would like to consider as being correct (from a practical point of view) turn out to be incorrect ones. This is illustrated by the *SET* specification given in Fig. 4.

1— A specification of sets of natural numbers

spec : SET ;
  use : NAT, BOOL ;
 sort : Set ;
 generated by :
  $\emptyset$: $\longrightarrow$ Set ;
  ins: Nat Set $\longrightarrow$ Set ;
 operations :
  _ $\in$ _ : Nat Set $\longrightarrow$ Bool ;
  del : Nat Set $\longrightarrow$ Set ;
 axioms :
  ins(x, ins(x, S)) = ins(x, S) ;
  ins(x, ins(y, S)) = ins(y, ins(x, S)) ;
  del(x, $\emptyset$) = $\emptyset$;
  del(x, ins(x, S)) = del(x, S) ;
  x $\neq$ y $\Longrightarrow$ del(x, ins(y, S)) = ins(y, del(x, S)) ;
  x $\in$ $\emptyset$ = false ;
  x $\in$ ins(x, S) = true ;
  x $\neq$ y $\Longrightarrow$ x $\in$ ins(y, S) = x $\in$ S ;
  where : S : Set ; x, y : Nat ;
 end SET .

If we consider a standard realization of *SET* by e.g. lists, we do not obtain a correct realization: this is due to the axioms expressing the commutativity of the insertion operation, which do not hold for lists. However, if we notice that indeed we are only interested in the result of some computations (e.g. membership), then it is clear that our realization of *SET* by lists "behaves" correctly. Thus, an intuitively correct realization of an algebraic specification *SP* may correspond to an algebra which is **not** in *Mod*(*SP*). This leads

to a refined understanding of software correctness: a program P should be considered as being correct w.r.t. its specification SP if and only if the algebra defined by P is an "observationally correct realization" of SP. In other words, the differences between the specification and the software should not be "observable", w.r.t. some appropriate notion of "observability".

The problem is now to specify the "observations" to be associated to some specification, and to define the semantics of such "observations" in order to obtain a framework that will capture the essence of software correctness. Up to now, various notions of observability have been introduced, involving observation techniques based on sorts [GGM76], [Wan79], [Kam83], [Gan83], [Rei85], [MG85], [Sch87], [NO87], [MT89], on operations [Die88], on terms [ST88], [Hen89] or on formulas [ST85], [ST88]. Assuming that we have chosen some observation technique, we can specify, using this technique, that some parts of an algebraic specification are observable. An observational specification is thus obtained by adding a specification of the objects to be observed to a usual algebraic specification. The next step is to define the semantics of these observational specifications, in such a way that our assertion "the class of the models of some specification represents all its acceptable realizations" is correctly reflected. As explained above, some correct software could correspond to an algebra which does not satisfy all the axioms of the specification, provided that the differences between the properties of the algebra and the properties required by the specification are not observable.

There are mainly two possible ways to define the semantics of observational specifications. We can extend the class of the models of the specification SP by including some additional algebras which are "behaviourally equivalent" (w.r.t. the specified observations) to a model of $Mod(SP)$ (*extension by behavioural equivalence*, see [ST85], [ST88], [Hen89]). In the sequel such an approach will be referred to as **behavioural semantics**. We can also directly relax the satisfaction relation, hence redefine $Mod(SP)$ (*extension by relaxing the satisfaction relation*, see [Rei85], [NO87], [Die88]). We will call these approaches **observational semantics**.

For a comparative study of these various ways of defining the semantics of observational specifications, and of the relative expressive power of the various observation techniques mentioned above, see [BBK91]. In the sequel we will provide a short insight into the behavioural approach, and we will point out some of its limitations. In the next section we will describe a semantics based on the observational approach.

To define a behavioural semantics we first need to define an appropriate equivalence relation $\equiv_{Obs}$ on the class $Mod(\Sigma)$ of all Σ-algebras, also called behavioural equivalence of algebras w.r.t. the specified observations Obs [ST85, ST88]. The definition of $\equiv_{Obs}$ depends on the observational technique in use (i.e. whether we observe sorts, operations, terms or formulas [BBK91]). Assuming that we observe a set of formulas Φ (which is the most general case), the behavioural equivalence $\equiv_{\Phi}$ and the associated behavioural semantics are defined as follows:

Definition 3 : (Behavioural semantics) [ST88]

Given a set of observed formulas Φ, the behavioural equivalence w.r.t. Φ, written

$\equiv_\Phi$, is an equivalence relation on $Mod(\Sigma)$ defined by:

$$A \equiv_\Phi B \quad \textit{if and only if} \quad \forall \varphi \in \Phi \quad A \models \varphi \quad \Leftrightarrow \quad B \models \varphi$$

In other words, two Σ-algebras A and B are behaviourally equivalent w.r.t. a set of observable formulas Φ, if and only if A and B satisfy the same observable formulas. The class of the behavioural models of some specification SP (with observed formulas Φ), written $Beh(SP, \Phi)$, is defined by:

$$Beh(SP, \Phi) = \{B \in Mod(\Sigma) \ \mid \ \exists\ A \in Mod(SP)\ s.t.\ B \equiv_\Phi A\}$$

Now we would like to point out some limitations intrinsic to behavioural semantics. It turns out that in some cases, behavioural semantics is not powerful enough to fully capture our requirements w.r.t. software correctness: in these cases, we know of some realization that we would like to consider as being correct, but unfortunately this realization cannot be shown to be behaviourally equivalent to any of the (usual) models of the specification at hand. A typical example of such cases arises when $Mod(SP)$ is empty (i.e. when the specification is inconsistent in the usual sense). For instance, let us consider a variant of our *SET* specification as described in Fig. 4.

What we really need for this example is to observe the following set of terms:

$$W \;=\; \{x \in S\} \cup \{t \in T_{LIST\text{-}signature}(X) \mid t \textit{ is of sort Nat or Bool }\}$$

In other words, we observe membership and some *LIST* terms but we do not observe those *LIST* terms where *enum* occurs.[6]

Obviously, the specification *SET-WITH-ENUM* is inconsistent (i.e. $Mod(SP) = \emptyset$). Consequently its class of behavioural models is empty as well, whatever the observations specified and the behavioural equivalence used. Nevertheless, a realization which represents sets by non redundant lists, *ins* being realized by *cons* (when the element to be inserted is not already in the list) and *enum* being a coercion, should clearly be considered as a correct one.

The point here is that in a behavioural approach, the existence of behavioural models depends on the existence of usual models. Indeed, behavioural semantics still rely on the usual satisfaction relation, hence behavioural consistency coincides with standard consistency. This is the reason why we shall develop in the next section an approach based on observational semantics, i.e. an approach where the satisfaction relation is redefined accordingly to the specified observations.

5 Observational specifications

In this section we develop a new framework for observational specifications, the semantics of which is based on a redefinition of the satisfaction relation. We will only consider flat or

[6]Note that for this example we observe terms and not formulas. However, as shown in [BBK91], behavioural equivalence can be defined in a similar way as above. Moreover, whatever the definition of $\equiv_{Obs}$ is, our counter-example remains.

2— A variant of the specification of sets of natural numbers
spec : SET-WITH-ENUM ;
  use : LIST, NAT, BOOL ;
 sort : Set ;
 generated by :
  $\emptyset$: $\longrightarrow$ Set ;
  ins: Nat Set $\longrightarrow$ Set ;
 operations :
  _ $\in$ _ : Nat Set $\longrightarrow$ Bool ;
  del : Nat Set $\longrightarrow$ Set ;
  enum : Set $\longrightarrow$ List ;
 axioms :
  ins(x, ins(x, S)) = ins(x, S) ;
  ins(x, ins(y, S)) = ins(y, ins(x, S)) ;
  del(x, $\emptyset$) = $\emptyset$;
  del(x, ins(x, S)) = del(x, S) ;
  x $\neq$ y $\Longrightarrow$ del(x, ins(y, S)) = ins(y, del(x, S)) ;
  x $\in \emptyset$ = false ;
  x $\in$ ins(x, S) = true ;
  x $\neq$ y $\Longrightarrow$ x $\in$ ins(y, S) = x $\in$ S ;
  enum($\emptyset$) = nil ;
  x $\in$ S = true $\Longrightarrow$ enum(ins(x, S)) = enum(S) ;
  x $\in$ S = false $\Longrightarrow$ enum(ins(x, S)) = cons(x, enum(S)) ;
  where : S : Set ; x, y : Nat ;
 end SET-WITH-ENUM .

structured observational specifications, modular ones being dealt with in the next section.

As explained in the previous section, we want to reflect the following idea: some data structures are observable with respect to some observable sorts (e.g. lists are observable w.r.t. their elements via terms such as $car(L)$ or $car(cdr(L))$ etc.), but we need also to prevent from observing the results of some specific operations (e.g. if a list is obtained by enumeration of a set, $enum(S)$, it must not be observed; in particular, $car(enum(S))$, which denotes a value of an observable sort, must nevertheless be non observable). This leads to the following idea: given a specification *SP*, one defines the set of *observable sorts* S_{Obs}, and in addition one defines the set of "operations allowing observations" which is a subset Σ_{Obs} of the signature of *SP* (e.g. Σ_{Obs} can contain all the operations except *enum*).

5.1 Definitions

Let us first define the syntax of (flat) observational specifications.

Definition 4 : (Observational specifications)

- An *observation Obs* over a signature (S, Σ) is a couple (S_{Obs}, Σ_{Obs}) such that $S_{Obs} \subset S$ and $\Sigma_{Obs} \subset \Sigma$.
- An *observational signature* is a couple (Σ, Obs) such that *Obs* is an observation over Σ.
- An *axiom* over a signature Σ is a sentence whose atoms are equalities (between two Σ-terms of the same sort, with variables) and whose connectives belong to $\{ \neg, \wedge, \vee, \Rightarrow \}$. Every variable is implicitly universally quantified.
- An *observational specification* is a couple *SP–Obs* $= (SP, Obs)$ such that $SP = (S, \Sigma, \mathcal{A}x)$ is a classical specification (i.e. $\mathcal{A}x$ is a set of axioms over Σ) and *Obs* is an observation over the signature Σ.
- If $\mathcal{A}x$ only contains equalities then *SP–Obs* is called *equational*. If $\mathcal{A}x$ only contains axioms of the form

  $$(u_1 = v_1) \wedge \ldots \wedge (u_n = v_n) \implies (u = v)$$

  then *SP–Obs* is called *positive conditional*.

Example 5 : We have seen that, for the specification *SET-WITH-ENUM* given in Fig. 4, we need to observe only the terms of sort *Bool* or *Nat* where the operation *enum* does not occur. Thus, it is sufficient to declare $S_{Obs} = \{Bool, Nat\}$ and $\Sigma_{Obs} = \Sigma - \{enum\}$ in order to obtain the required set of observable terms W already mentioned in Section 4.

As usual, the notion of *observable contexts* is crucial for observability [Kam83, Rei85, NO87, Hen89, Hen90]:

Definition 6 : (Observable contexts)

- In general a *context* over a signature Σ is a Σ-term C with exactly one occurrence of one variable.
- Given a context C, its *arity* is $(s' \to s)$, where s' is the sort of the variable occurring in C and s is the sort of (the term) C. s is also called the (target) sort of C.
- Let (Σ, Obs) be an observational signature; the associated set of *observable contexts* is the set $\mathcal{C}_{Obs}$ which contains all the contexts over the signature Σ_{Obs} whose target sort belongs to S_{Obs}.
- For each observable sort $s \in S_{Obs}$, the context reduced to a variable of sort s is called "the empty context" (of sort s).

Let us now define the semantics of (flat) observational specifications.

Definition 7 : (Observational semantics)
Let *SP–Obs* be an observational specification and Σ be its signature. Let M be a Σ-algebra and let ax be an axiom of *SP–Obs*.

- Two elements a and b of M are *observationally equal* with respect to *Obs* if and only if they have the same sort s and for all contexts $C \in \mathcal{C}_{Obs}$ of arity $s \to s'$, $C(a) = C(b)$ in M (according to the usual equality of set theory). In particular observational equality on observable sorts coincides with the set-theoretic equality;[7] for the non observable sorts, the observational equality contains the set-theoretic equality, but there are also distinct values which are observationally equal.
- The algebra M *satisfies* ax with respect to *Obs* means that for all substitutions $\sigma : T_\Sigma(X) \to M$, $\sigma(ax)$ holds in M according to the observational equality (defined above) and the truth tables of the connectives.
- The algebra M *satisfies SP–Obs* means that it satisfies all the axioms of *SP–Obs* with respect to *Obs*.
- The satisfaction of observational equalities is denoted by "$\models_{Obs}$" and we write "$M \models_{Obs} (a = b)$", "$M \models_{Obs}$ *SP–Obs*"...

Example 8 : It is not difficult to show that the realization of *SET-WITH-ENUM* by non redundant lists described in the previous section satisfies the observational specification given above. For instance:

- When $x \notin S$, $enum(insert(x, S)) = cons(x, enum(S))$ is satisfied because they are equal (with respect to the set-theoretic equality) in our model; thus, they are a fortiori observationally equal.
- $insert(x, insert(y, S)) = insert(y, insert(x, S))$ is observationally satisfied (even when these two list realizations of sets are not equal with respect to the set-theoretic equality) because all contexts involving *enum* do not belong to $\mathcal{C}_{Obs}$; here, all the observable contexts C in $\mathcal{C}_{Obs}$ have $Set \to Bool$ as arity and the top symbol of C is necessarily "$\in$".

[7] because $\mathcal{C}_{Obs}$ always contains the empty contexts on observable sorts.

Notation 9 : Given an observational specification *SP–Obs*, *Mod*(*SP–Obs*) is the full sub-category of *Mod*(Σ) whose objects are the Σ-algebras satisfying *SP–Obs*.

The following results are trivial:

Fact 10 : Given an observational signature (Σ, Obs), Σ-morphisms preserve observational equalities: for all $\mu : M \rightarrow M'$, if $M \models_{Obs} (a = b)$ then $M' \models_{Obs} (\mu(a) = \mu(b))$.

Fact 11 : *Mod*(*SP*) is equal to *Mod*(*SP–Obs*) when $S_{Obs} = S$ (due to the empty contexts).

Fact 12 : If *SP–Obs* is equational then the usual category *Mod*(*SP*) is a full sub-category of *Mod*(*SP–Obs*).

Fact 13 : More generally, if *SP* is an equational specification and if $Obs_1 \subset Obs_2$ then $Mod(SP\text{–}Obs_2)$ is a full sub-category of $Mod(SP\text{–}Obs_1)$.

Notice that, from Fact 11, Fact 12 is a particular case of Fact 13. Moreover, the inclusions stated in Fact 12 (hence in Fact 13) are often strict: there is often a model M with two elements $a \neq b$ such that $M \models_{Obs} (a = b)$.

Fact 14 : Fact 12, hence Fact 13, cannot be extended to non equational specifications. For example let M be an algebra such that $a \neq b$ and $c \neq d$ with $M \models_{Obs} (a = b)$ and $M \not\models_{Obs} (c = d)$. Then, M satisfies $[(a = b) \Rightarrow (c = d)]$ in the classical sense because the precondition is false, but it does not satisfy this axiom with respect to *Obs*.

Our specification of *SET-WITH-ENUM*[8] is an example where *Mod*(*SP*) (i.e. without observability) only contains algebras with a trivial *Nat* carrier (a singleton),[9] but *Mod*(*SP–Obs*) contains, among others, algebras where the *Nat* part is isomorphic to $\mathbb{N}$.

5.2 Initiality results

As usual, initiality results can only be easily obtained for equational, or positive conditional, specifications [WB80].

Theorem 15 : (Least congruence)
Let *SP–Obs* be a **positive conditional** observational specification and M be a Σ-algebra. There exists a least congruence $\equiv$ on M such that the quotient algebra $M/_{\equiv}$ satisfies *SP–Obs*.

[8] when flattened.

[9] because $car(enum(ins(n, ins(m, \emptyset)))) = car(enum(ins(m, ins(n, \emptyset))))$, hence $n = m$

Sketch of the proof: Let F be the family of all the congruences such that $M/_{\equiv}$ satisfies *SP–Obs*. It is not empty because the trivial congruence τ defined by $(a \; \tau \; b) \; \Leftrightarrow \; (a \textit{ and } b$ *have the same sort*) belongs to F. Let $\equiv$ be the intersection of all the congruences in F; if $\equiv$ still belongs to F then the theorem is proved. Consequently, we simply have to prove that $M/_{\equiv}$ satisfies (observationally) each axiom of *SP*. This is not difficult, by applying the definitions. □

The following corollary is simply obtained from the previous theorem with $M = T_\Sigma$ (as in [GTW78]):

Corolary 16 : (Initial object)
The category *Mod*(*SP–Obs*) has an initial object $I = (T_\Sigma/_{\equiv})$.

It may seem surprising that, regardless of several other works on observability [Kam83, MMG87, MT89], we care about initial objects while they care about terminal ones. Indeed we believe that any "collapse between values" reflects some **implementation choice** (each implementation choice being intuitively reflected by some equation which induces new collapses). From this viewpoint, "considering the least congruence" means "considering only the necessary implementation choices"; thus, the initial algebra can be considered as the most general realization. More generally, when the initial algebra does not exist, minimal models can be considered as realizations with "minimal implementation choices". Moreover, *Mod*(*SP–Obs*) has always a terminal object which is the trivial algebra. This algebra has clearly no interest. This is due to the fact that, for the moment, our specifications are **flat**. Terminal algebras get interest only when some enriched specifications are "protected". We shall consider such hierarchical constraints when observational semantics and the stratified loose approach will be integrated together.

5.3 Structured observational specifications

The following proposition generalizes Fact 13 above.

Proposition 17 : Let $SP\text{–}Obs_1$ and $SP\text{–}Obs_2$ be two observational specifications such that $SP\text{–}Obs_1 \subset SP\text{–}Obs_2$. If $SP\text{–}Obs_1$ is equational then the forgetful functor $\mathcal{U}$ from $Mod(\Sigma_2)$ to $Mod(\Sigma_1)$ has the following property:
For all Σ_2-algebras M satisfying $SP\text{–}Obs_2$, the Σ_1-algebra $\mathcal{U}(M)$ satisfies $SP\text{–}Obs_1$. Then $\mathcal{U}$ also denotes the forgetful functor from $Mod(SP\text{–}Obs_2)$ to $Mod(SP\text{–}Obs_1)$.

Proof : Results from $\mathcal{C}_{Obs-1} \subset \mathcal{C}_{Obs-2}$. □

This proposition can be extended to positive conditional observational specifications whose axioms only contains equalities of **observable sorts** (in S_{Obs}) **in the preconditions**. Such an extension is similar to some sufficient conditions used in [Hen90, BGM91] for proof methods with observability.

Note that, from Fact 14 above, this proposition cannot in general be extended to a non equational specification $SP\text{–}Obs_1$. Moreover, for the same reason, observational

specifications do not define an institution [GB84] because the *Satisfaction Condition* is not guaranteed in our framework. Anyway, it seems clear that this counter-fact is intrinsic to the observability question: there is no reasonable syntactical constraint which ensures that an enrichment does not add new observations of old values. Consequently some axioms which were satisfied by the models of a specification can become unsatisfied when new observations are added. This can only be handled through modularity constraints, which cannot be easily reflected within the institution framework (just because of the *Satisfaction Condition*). As explained later on, we shall reflect these constraints owing to the stratified loose semantics.

Provided that the forgetful functor $\mathcal{U} : Mod(SP\text{–}Obs_2) \rightarrow Mod(SP\text{–}Obs_1)$ exists, and that $SP\text{–}Obs_2$ is positive conditional, $\mathcal{U}$ as a left adjoint, as stated in the following theorem:

Theorem 18 : (Free synthesis functor)

Let $SP\text{–}Obs_1$ and $SP\text{–}Obs_2$ be two observational specifications such that $SP\text{–}Obs_1 \subset SP\text{–}Obs_2$. If $SP\text{–}Obs_1$ is equational and $SP\text{–}Obs_2$ is positive conditional then the forgetful functor $\mathcal{U}$ admits a left adjoint functor $\mathcal{F}_\Delta$ from $Mod(SP\text{–}Obs_1)$ to $Mod(SP\text{–}Obs_2)$. In particular, there is a unit of adjunction which provides a canonical Σ_1-morphism from M to $\mathcal{U}(\mathcal{F}_\Delta(M))$ for every algebra M in $Mod(SP\text{–}Obs_1)$.

Sketch of the proof: We use the existence of a minimal congruence exactly as for the classical ADJ framework of algebraic specifications with positive conditional axioms. For each $M \in Mod(SP\text{–}Obs_1)$ we consider the Σ_2-algebra $T_{\Sigma_2}[M]$. Let $\equiv$ be the least congruence on $T_{\Sigma_2}[M]$ generated by the (observational) axioms of $SP\text{–}Obs_2$ and the fibers of the canonical morphism from $T_{\Sigma_1}[M]$ to M. The $SP\text{–}Obs_2$ algebra $T_{\Sigma_2}[M]/_\equiv$ is by definition $\mathcal{F}_\Delta(M)$. It is not difficult (but tedious!) to prove that $\mathcal{F}_\Delta$ is compatible with the morphisms (thus it is a functor) and that there is a natural bijection between $Hom_{SP\text{–}Obs_1}(X, \mathcal{U}(Y))$ and $Hom_{SP\text{–}Obs_2}(\mathcal{F}_\Delta(X), Y)$ for all objects $X \in Mod(SP\text{–}Obs_1)$ and $Y \in Mod(SP\text{–}Obs_2)$ (which is the definition of adjunction). □

As usual, the existence of the unit of adjunction allows to define *hierarchical consistency* within an initial framework [Ber87].

Definition 19 : (Hierarchical consistency)

Let $SP\text{–}Obs_1$ and $SP\text{–}Obs_2$ be two observational specifications such that $SP\text{–}Obs_1 \subset SP\text{–}Obs_2$, $SP\text{–}Obs_1$ is equational and $SP\text{–}Obs_2$ is positive conditional. The observational specification $SP\text{–}Obs_2$ is *hierarchically consistent* w.r.t. $SP\text{–}Obs_1$ if and only if the canonical morphism from I_1 to $\mathcal{U}(I_2)$ is a monomorphism (i.e. is injective in our framework), where I_1 (resp. I_2) denotes the initial algebra of $Mod(SP\text{–}Obs_1)$ (resp. $Mod(SP\text{–}Obs_2)$).

Remember that left adjoint functors preserve initial models, thus $I_2 = \mathcal{F}_\Delta(I_1)$.

Unfortunately, a similar definition of *sufficient completeness* (via the surjectivity of the canonical morphism from I_1 to $\mathcal{U}(I_2)$) is not adequate. For example, when considering our *SET-WITH-ENUM* enrichment, this canonical morphism is not an epimorphism: in the

initial object I_2, the terms $enum(S)$ create new list values which are only **observationally** equal to old list values. Thus, the following definition could be better:

Definition 20 : (Sufficient completeness)
Let $SP\text{–}Obs_1$ and $SP\text{–}Obs_2$ be two observational specifications such that $SP\text{–}Obs_1 \subset SP\text{–}Obs_2$, $SP\text{–}Obs_1$ is equational and $SP\text{–}Obs_2$ is positive conditional. The observational specification $SP\text{–}Obs_2$ is *sufficiently complete* w.r.t. $SP\text{–}Obs_1$ if and only if the canonical morphism μ from I_1 to $\mathcal{U}(I_2)$ has the following property:
For all values $v \in \mathcal{U}(I_2)$, there exists a value $u \in I_1$ such that $\mathcal{U}(I_2) \models_{Obs} (v = \mu(u))$.

This allows us to define *persistency*:

Definition 21 : (Persistency)
Let $SP\text{–}Obs_1$ and $SP\text{–}Obs_2$ be two observational specifications such that $SP\text{–}Obs_1 \subset SP\text{–}Obs_2$, $SP\text{–}Obs_1$ is equational and $SP\text{–}Obs_2$ is positive conditional. The observational specification $SP\text{–}Obs_2$ is *persistent* w.r.t. $SP\text{–}Obs_1$ if and only if it is hierarchically consistent and sufficiently complete.

Of course, such an initial approach is rather restrictive. We must more or less restrict ourselves to equational specifications in order to exploit the results stated in this section. However, almost all the classical results build on the top of the ADJ group approach are then usable. In particular, if a specification has been written following the "fair presentation" method then it is sufficiently complete, and if there are no explicit equations between generators then it is persistent [Bid82].

Nevertheless, we believe that our definition of sufficient completeness is not fully satisfactory because I_2 does not protect the predefined data structure reflected by I_1. Indeed, our realization of sets by non redundant lists already described is a suitable model, while I_2 is not a suitable model. This means that the initial approach is not fully adequate: hierarchical constraints should be substituted to sufficient completeness. More generally, structured specifications are probably not powerful enough to capture the essence of observability. In other words, we believe that observability issues intrinsically require a modular approach with semantic constraints.

6 Modular observational specifications

In this section we show how we can obtain a satisfactory approach to software correctness by embedding observability into the stratified loose approach defined in Section 3.

Remember that when we have defined the stratified loose semantics of modular specifications in Section 3, we have claimed that this definition was (more than) institution independent. We will benefit here from this property, since considering modular observational specifications (with the observational semantics defined in the previous section) instead of standard modular specifications directly provides us with the adequate semantics we are looking for. To be more precise, we have explained in the previous section that

the observational semantics we have introduced does not lead to an institution of observational specifications. However, since the existence of forgetful functors from Σ_2-algebras to Σ_1-algebras is the only requirement really needed for the stratified loose approach, there is no difficulty to translate the definition of the stratified loose semantics for modular observational specifications.

Thus, combining the stratified loose semantics (for modularity) with the observational semantics defined in Section 5 provides a framework where:

- The global correctness of some software w.r.t. its formal specification can be established on a module per module basis.
- Local correctness is defined in a way flexible enough to cope with "non observable" differences between the properties of the software module and the properties specified by the corresponding specification module.

The crucial point here is that the hierarchical constraints induced by the stratified loose semantics will guarantee us that the composition of correct software modules will always result in a correct software, and that the various modules of the specification can be implemented independently of each other. Hence we do not have to worry about the somewhat problematic features discussed at the end of Section 5. More precisely, the "no junk" and "no confusion" properties inherent to the stratified loose approach (cf. Section 3) are still valid here, and there is no need for the definitions of "sufficient completeness" and "hierarchical consistency" given in the initial approach to structured observational specifications. Moreover, in the previous section we have provided arguments for demonstrating that our observational semantics is powerful (i.e. "flexible") enough, since the counter example discussed at the end of Section 4 was solved in an elegant way by an adequate redefinition of the satisfaction relation.

We believe that the framework developed in this paper provides a firm basis to establish the correctness of some (modular) software w.r.t. its (modular, observational) specification. However, if we really want to prove the correctness of some software, then we need adequate deduction rules and proof techniques. This point is far beyond the scope of this paper, but we would nevertheless discuss some proof related aspects. Remember that in Section 3 we have introduced the restriction to finitely generated models (w.r.t. the operations specified as generators) to guarantee that "induction w.r.t. the generators" is a correct proof principle. An obvious question is whether a similar restriction can be introduced in the framework of observational specifications, and whether we will obtain a similar powerful proof principle.

As a first remark we should note that the restriction to finitely generated models (w.r.t. the generators) is not adequate since such a restriction will be somehow contradictory with the aim of the observational semantics we have developed so far. To illustrate this we will consider the following example:

Example 22 : (Stacks implemented by arrays)
Let us consider an observational specification of stacks of natural numbers where S_{Obs} is the singleton $\{Nat\}$ and Σ_{Obs} is equal to Σ; the generators being obviously

emptystack and *push* for the sort *Stack*. Of course, we would like to consider a model which implements stacks by means of arrays as an observationally correct model: stack values are couples (a, h) where a is an array and h is the height of the stack; *emptystack* is realized by some initial array $a = init$ and $h = 0$, *push* records its element at range h in a and increments h, *pop* simply decreases h without modifying a, etc.

Let us assume that the initial array *init* uniformly contains 0 for all indices. Then, all stacks values obtained via the generators *emptystack* and *push* satisfy the following property: for all indices $i \geq h$, $a[i] = 0$. But this property is not satisfied for the stack $pop(push(1, emptystack))$ (because $h = 0$ and $a[0] = 1$ for this stack value). Consequently this model is not finitely generated w.r.t. the generators *emptystack* and *push*.

Nevertheless, one should remark that even though $pop(push(1, emptystack))$ is not equal to *emptystack* according to the set-theoretic equality in our model, it is **observationally equal** to *emptystack* .

Thus, it is clear that we must allow values that are not denotable by a composition of generators; but we can still obtain the desired proof principle by requiring for each value to be observationally equal to a value denotable by a composition of generators. This leads to the following definition.

Definition 23 : (Observational restriction to generators)
Let *SP–Obs* be a modular observational specification. Let $\Omega \subset \Sigma$ be the set of generators declared in *SP–Obs*. A model M of *SP–Obs* is *observationally finitely generated w.r.t.* Ω if and only if for every value m in M there exists an Ω-term t such that $M \models_{Obs} (m = t)$.

As a second remark, we would like to point out that refining the stratified loose semantics with the "observational restriction to generators" constraint has at least two advantages. It simplifies the proof principles and moreover, it has an important consequence on the "specification style": some operations can be specified in a really abstract manner, as demonstrated in the following toy example:

Example 24 : (pickout in sets)
Let us consider a specification of sets with a *pickout* operation which is supposed to delete **one of** the elements of a set. We do not want to specify **which** element has to be deleted. This specification module is described in Fig. 24.

For sake of simplicity, let us assume that the elements are the boolean values. The models that we clearly would like to accept are the ones which contain four set values up to observational equality: $\emptyset$, $\{true\}$, $\{false\}$ and $\{true, false\}$. Two possible behaviours of *pickout* are acceptable: $pickout(\{true, false\}) = \{true\}$ or $pickout(\{true, false\}) = \{false\}$.[10] As a matter of fact, we exactly get these models when we consider the semantic constraint of "observational restriction to generators". If this constraint is not required, then we get exotic models such as:

[10] These equalities are only **observational** ones.

3— Yet another specification of sets

spec : SET-WITH-PICKOUT ;
use : ELEM, NAT, BOOL ;
sort : Set ;
generated by :
$\emptyset : \longrightarrow$ Set ;
ins: Elem Set $\longrightarrow$ Set ;
operations :
$_ \in _$: Elem Set $\longrightarrow$ Bool ;
card : Set $\longrightarrow$ Nat ;
pickout : Set $\longrightarrow$ Set ;
axioms :
$\text{ins}(x, \text{ins}(x, S)) = \text{ins}(x, S)$;
$\text{ins}(x, \text{ins}(y, S)) = \text{ins}(y, \text{ins}(x, S))$;
$x \in \emptyset = \text{false}$;
$x \in \text{ins}(x, S) = \text{true}$;
$x \neq y \Longrightarrow x \in \text{ins}(y, S) = x \in S$;
$\text{card}(\emptyset) = 0$;
$x \in S = \text{false} \Longrightarrow \text{card}(\text{ins}(x, S)) = \text{succ}(\text{card}(S))$;
$\text{pickout}(\emptyset) = \emptyset$;
$S \neq \emptyset \Longrightarrow \text{card}(\text{pickout}(S)) = \text{pred}(\text{card}(S))$;
$x \in \text{pickout}(S) = \text{true} \Longrightarrow x \in S = \text{true}$;
where : S : Set ; x, y : Elem ;
end SET-WITH-PICKOUT .

- The set carrier contains five values: $\emptyset$, $\{true\}$, $\{false\}$, $\{true, false\}$ and a strange set $\{true{+}false\}$.
- *ins* is a constant function on $\{true{+}false\}$ which always returns $\{true{+}false\}$ itself and it works as usual on the other sets.
- $_ \in _$ is the constant function *true* on $\{true{+}false\}$ and it works as usual on the other sets.
- $card(\{true{+}false\}) = 1$ and the cardinal of the other sets is the usual one.
- $pickout(\{true, false\}) = \{true{+}false\}$ and *pickout* on all other sets returns the empty set.

This model clearly satisfies the specification. However $\{true{+}false\}$ is not observationally equal to a standard set because there are two distinct values (*true* and *false*) which are members of it, but its cardinal is 1.

This example, together with the *remove* and *div* examples given in Section 3, show that semantic constraints are fundamental in order to reach a specification style which is really **abstract**.

Obviously, the definition of correct proof principles for modular observational specifications requires further investigation. Some combination of "observational induction w.r.t. the generators" and of "context induction" à la Hennicker [Hen90] could prove adequate.

As a last remark, we would like to remind that, as for standard modular specifications, the hierarchical inconsistency of a given observational specification module can result either from "inconsistent axioms" or from the introduction of "new" observations on "old" sorts. However, in the framework of modular observational specifications, "inconsistent axioms" and "adding new observations on old sorts" should be interpreted with respect to the observational satisfaction relation and the specified observations ΔObs. It is clear that the "flexibility" induced by observability will prove useful for hierarchical consistency issues as well. Moreover, if it is obvious that the observations should be carefully designed, this task is made easier since they are explicitly specified.

7 Specifying adequate observations

In this section we would like to hark back to our claim that the observational semantics defined in Section 5 is powerful ("flexible") enough, and to the role of those operations who prevent some observations (such as *enum*).

Our *SET-WITH-ENUM* example (cf. Fig. 4) was used to justify the need for an observational semantics. However, one could argue that this example was a bit ad hoc, since the purpose of the *enum* operation was rather mysterious: what could be the use of an operation which never provides observable results?

In general, such operations correspond to "internal services" used to define some other operations. For instance, assume that the *LIST* module provides a *sum* operation, which computes the sum of all the natural numbers contained in a list. Assume moreover that

this *sum* operation belongs to Σ_{Obs}. Then we can compute the sum of all the natural numbers contained in a set by the following term: $sum(enum(S))$.

Well, things are not that easy: the term $sum(enum(S))$ is not observable. Nevertheless, this apparent difficulty can be easily solved by defining a new operation $sigma : Set \rightarrow Nat$, in the *SET-WITH-ENUM* specification module, with the following axiom: $sigma(S) = sum(enum(S))$. It is then sufficient to specify that *sigma* belongs to Σ_{Obs} and we are done. Note that the resulting specification module remains hierarchically consistent, since the *sum* operation on lists is associative and commutative.

One could believe that the need for a new operation *sigma* exhibits some weakness of our approach. On the contrary, our point is that preventing from observing the results of some operations can be considered as some kind of a very flexible visibility control mechanism. More precisely, these operations are "internal services" who can be used to define more complex computations, and as such they are available through the whole specification. However, a "client" of any realization of the specification is not allowed to directly invoke these internal services (e.g. by the term $sum(enum(S))$), but should instead invoke some explicitly made available service (e.g. *sigma*).

8 Conclusion

We have investigated how far modularity and observability issues can contribute to a better understanding of software correctness. We have detailed the impact of modularity on the semantics of algebraic specifications. We have shown that, with the stratified loose semantics, software correctness can be established on a module per module basis. Then we have discussed observability issues. In particular, we have explained why a behavioural semantics of observability (based on an equivalence relation between algebras) is not fully satisfactory. Therefore, we have introduced an observational semantics (based on a redefinition of the satisfaction relation) where sort observation is refined by specifying that some operations do not allow observations. Then we have integrated the stratified loose approach and our observational semantics together. As a result, we have obtained a framework (modular observational specifications) where the definition of software correctness is adequate, i.e. fits with actual software correctness. Moreover, we have shown that, with modular observational specifications, we reach a specification style which is really abstract. Our definition of software correctness is a first step towards putting software correctness proofs in practice. A promising area for further investigations is the development of (modular) proof methods on top of our approach.

Acknowledgement: We would like to thank Teodor Knapik for numerous fruitful discussions. This work is partially supported by C.N.R.S. GRECO de Programmation and E.E.C. Working Group COMPASS.

References

[AW86] E. Astesiano and M. Wirsing. : "*An introduction to ASL.*" In Proc. of the IFIP WG2.1 Working Conference on Program Specifications and Transformations, 1986.

[BBC86] G. Bernot, M. Bidoit, and C. Choppy. : "*Abstract implementations and correctness proofs.*" In Proc. of the 3rd Symposium on Theoretical Aspects of Computer Science (STACS), pages 236–251, Springer-Verlag L.N.C.S. 210, 1986.

[BBK91] G. Bernot, M. Bidoit, and T. Knapik. : "*Observational approaches in algebraic specifications: A comparative study.*" Technical Report 6, LIENS, 1991.

[Ber87] G. Bernot. : "*Good functors... are those preserving philosophy.*" In Proc. of the Summer Conference on Category Theory and Computer Science, pages 182–195, Springer-Verlag L.N.C.S. 283, 1987.

[BGM89] M. Bidoit, M.-C. Gaudel, and A. Mauboussin. : "*How to make algebraic specifications more understandable? an experiment with the Pluss specification language.*" Science of Computer Programming, 12(1), 1989.

[BGM91] G. Bernot, M.-C. Gaudel, and B. Marre. : "*Software testing based on formal specifications: A theory and a tool.*" Software Engineering Journal, 1991.

[Bid82] M. Bidoit. : "*Algebraic data types: Structured specifications and fair presentations.*" In Proc. of the AFCET Symposium on Mathematics for Computer Science, 1982.

[Bid87] M. Bidoit. : "*The stratified loose approach: A generalization of initial and loose semantics.*" In Recent Trends in Data Type Specification, Selected Papers of the 5th Workshop on Specifications of Abstract Data Types, pages 1–22, Springer-Verlag L.N.C.S. 332, 1987.

[Bid89] M. Bidoit. : "*Pluss, un langage pour le développement de spécifications algébriques modulaires.*" Thèse d'Etat, Université Paris-Sud, 1989.

[Bid91] M. Bidoit. : "*Development of modular specifications by stepwise refinements using the Pluss specification language.*" In Proc. of the Unified Computation Laboratory, Oxford University Press, 1991.

[CIP85] F.L. Bauer et al. : "*The Munich project CIP. Volume I: The wide spectrum language CIP-L.*" Springer-Verlag L.N.C.S. 183, 1985.

[Die88] N.W.P. van Dieppen. : "*Implementation of modular algebraic specifications.*" In Proc. of the European Symposium on Programming (ESOP), pages 64–78, Springer-Verlag L.N.C.S. 300, 1988.

[EFH83] H. Ehrig, W. Fey, and H. Hansen. : "*ACT ONE: an algebraic specification language with two levels of semantics.*" Technical Report 83–03, TU Berlin FB 20, 1983.

[EKMP82] H. Ehrig, H.-J. Kreowski, B. Mahr, and P. Padawitz. : "*Algebraic implementation of abstract data types.*" Theoretical Computer Science, 20:209–263, 1982.

[EM85] H. Ehrig and B. Mahr. : "*Fundamentals of algebraic specification 1. Equations and initial semantics.*" Volume 6 of EATCS Monographs on Theoretical Computer Science, Springer-Verlag, 1985.

[FGJM85] K. Futatsugi, J.A. Goguen, J.-P. Jouannaud, and J. Meseguer. : "*Principles of OBJ2.*" In Proc. of the 12th ACM Symposium on Principles of Programming Languages (POPL), pages 52–66, 1985.

[Gan83] H. Ganzinger. : "*Parameterized specifications: Parameter passing and implementation with respect to observability.*" ACM Transactions on Programming Languages and Systems, 5(3):318–354, 1983.

[GB84] J.A. Goguen and R.M. Burstall. : "*Introducing institutions.*" In Proc. of the Workshop on Logics of Programming, pages 221–256, Springer-Verlag L.N.C.S. 164, 1984.

[GGM76] V. Girratana, F. Gimona, and U. Montanari. : "*Observability concepts in abstract data type specification.*" In Proc. of Mathematical Foundations of Computer Science (MFCS), pages 576–587, Springer-Verlag L.N.C.S. 45, 1976.

[GHW85] J.V. Guttag, J.J. Horning, and J.M. Wing. : "*Larch in five easy pieces.*" Technical Report 5, Digital Systems Research Center, 1985.

[GM88] M.-C. Gaudel and T. Moineau. : "*A theory of software reusability.*" In Proc. of the European Symposium on Programming (ESOP), pages 115–130, Springer-Verlag L.N.C.S. 300, 1988.

[GTW78] J.A. Goguen, J.W. Thatcher, and E.G. Wagner. : "*An initial approach to the specification, correctness, and implementation of abstract data types.*" Volume 4 of Current Trends in Programming Methodology, Prentice Hall, 1978.

[Hen89] R. Hennicker. : "*Implementation of parameterized observational specifications.*" In Proc. of TAPSOFT, pages 290–305, Springer-Verlag L.N.C.S. 351, 1989.

[Hen90] R. Hennicker. : "*Context induction: A proof principle for behavioural abstractions and algebraic implementations.*" Technical Report MIP–9001, Fakultät für Mathematik und Informatik, Universität Passau, 1990.

[Kam83] S. Kamin. : "*Final data types and their specification.*" ACM Transactions on Programming Languages and Systems, 5(1):97–123, 1983.

[McL71] S. Mac Lane. : "*Categories for the working mathematician.*" Volume 5 of Graduate Texts in Mathematics, Springer-Verlag, 1971.

[MG85] J. Meseguer and J.A. Goguen. : "*Initiality, induction and computability, pages 459–540.*" Algebraic Methods in Semantics, Cambridge University Press, 1985.

[MMG87] L.S. Moss, J. Meseguer, and J.A. Goguen. : "*Final algebras, cosemicomputable algebras and degrees of unsolvability.*" In Proc. of Category Theory and Computer Science, pages 158–181, Springer-Verlag L.N.C.S. 283, 1987.

[MT89] L.S. Moss and S.R. Thate. : "*Generalization of final algebra semantics by relativization.*" In Proc. of the 5th Mathematical Foundations of Programming Semantics International Conference, pages 284–300, Springer-Verlag L.N.C.S. 442, 1989.

[NO87] P. Nivela and F. Orejas. : "*Initial behaviour semantics for algebraic specification.*" In Recent Trends in Data Type Specification, Selected Papers of the 5th Workshop on Specification of Abstract Data Types, pages 184–207, Springer-Verlag L.N.C.S. 332, 1987.

[Rei85] H. Reichel. : "*Behavioural validity of conditional equations in abstract data types.*" In Contributions to General Algebra 3, Proc. of the Vienna Conference, 1984.

[Sch87] Oliver Schoett. : "*Data abstraction and the correctness of modular programming.*" PhD thesis, University of Edinburg, 1987.

[ST84] D.T. Sannella and A. Tarlecki. : "*Building specifications in an arbitrary institution.*" In Proc. of the International Symposium on Semantics of Data Types, Springer-Verlag L.N.C.S. 173, 1984.

[ST85] D. Sannella and A. Tarlecki. : "*On observational equivalence and algebraic specification.*" In Proc. of TAPSOFT, pages 308–322, Springer-Verlag L.N.C.S. 185, 1985.

[ST88] D. Sannella and A. Tarlecki. : "*Toward formal development of programs from algebraic specification revisited.*" Acta Informatica, (25):233–281, 1988.

[Wan79] M. Wand. : "*Final algebra semantics and data type extensions.*" Journal of Computer and System Sciences, 19:27–44, 1979.

[WB80] M. Wirsing and M. Broy. : "*Abstract data types as lattices of finitely generated models.*" In Proc. of the 9th Symposium on Mathematical Foundations of Computer Science (MFCS), 1980.

[Wir83] M. Wirsing. : "*Structured Algebraic specifications: A kernel language.*" PhD thesis, Techn. Univ. Munchen, 1983.

Chapitre 5 :
Label algebras and exception handling

Gilles BERNOT
LIENS, CNRS URA 1327
Ecole Normale Supérieure
45 Rue d'Ulm
F-75230 PARIS Cédex 05
FRANCE
bitnet: bernot@frulm63
uucp: bernot@ens.ens.fr

Pascale LE GALL
LRI, UA CNRS 410
Université PARIS-SUD
Bât. 490
F-91405 Orsay cedex
FRANCE
bitnet: legall@frlri61
uucp: legall@lri.lri.fr

Contents

1 Introduction **171**

2 Crucial aspects of exception handling **172**

2.1 Exception handling and programming languages 172

2.2 Exception handling and formal specifications 173

3 Algebraic specifications with exception handling **174**

3.1 Errors as constant operations 174

3.2 Errors and *OK* predicates 175

3.3 Errors and partial functions 176

3.4 Error handling and subsorting 176

3.5 Safe and unsafe operations 178

3.6 Bounded data structures and recovery axioms 179

3.7 About values, terms and labels 181

4 Label algebras **184**

4.1 Basic definitions 184

4.2 Some applications of label algebras 186

4.3 The partial evaluation constraint 188

5 Fundamental results **191**

6 Exception algebras **196**

6.1 Label algebras and exception algebras 196

6.2 Exception signature 198

6.3 Exceptions and errors 201

7 Exception specifications **203**

7.1 Generalized axioms 203

7.2 Ok-axioms 205

7.3 Constrained exception algebras 209

8 Main results and structured exception specifications **210**

8.1 Fundamental results 210

8.2 Structured exception specifications 211

9 Some examples **214**

9.1 Several versions for natural numbers 214

9.2 Other examples 217

10 Conclusion **222**

1 Introduction

Exception handling is a subject which, in practice, is often neglected in software engineering, especially at the specification stage. This results in incomplete specifications and the various choices of "how to treat exceptional cases" are then often made at the programming stage. As usual when specifications are incomplete, this decreases the overall quality of the software: some exceptional cases are checked twice (e.g. in the calling module and in the called module), or even worse there are misunderstandings about how they should be treated, or still worse they are never checked. Moreover, if the exceptional cases are not well specified, the corresponding bugs are very difficult to identify, as they do not cope with the standard verification and validation methods (e.g. proving or testing methods).

Nevertheless, most of these exceptional cases are fairly easy to classify. An important class of exceptional cases is related to "intrinsic" properties of the underlying abstract data structure: access to an empty data structure (e.g. top of an empty stack, or choosing an element in an empty set, etc.), or functions which are intrinsically not defined for certain values (e.g. popping an empty stack, predecessor for 0 in natural numbers, or factorial for negative numbers, etc.). Another important class of exceptional cases relies on "dynamic" properties of the data structure (e.g. access to a non-initialized data, non-initialized array cell, etc.). In addition, it is very important not to neglect certain limitations, due to the system itself or required by the specifier, mainly bounded data structures (e.g. arrays, intervals, etc.).

Over the last fifteen years [LZ75][Gut75][GTW78], *algebraic specifications* have been widely advocated as one of the most fruitful formal specification methods. In this paper, a new framework for exception handling within algebraic specifications is proposed. Before defining what we call *exception algebras*, we will introduce a general framework, the *label algebras*, whose application domain seems to be much more general than exception handling.

- In Section 2 we will point out two crucial aspects of exception handling that are often neglected: *clarity* and *terseness*. We will start from an example of algorithm which is considerably simplified when the programming language has exception handling features. By analogy, we will extract several requirements for formal specifications with exception handling in order to reach clarity and terseness.
- In Section 3 we will enumerate the main difficulties raised by exception handling within the algebraic framework (often resulting in inconsistencies). We will show that the most difficult point is to simultaneously handle bounded data structures and certain recoveries of exceptional values. Indeed, we will show that there was no existing framework capable of solving this difficulty. Solving this difficulty requires defining assignments on terms instead of values, and we will show the necessity of "labelling" terms in order to easily specify exceptions.
- In Section 4 we will define the framework of label algebras. We will also sketch out how far this framework can be applied to several other classical subjects of abstract data types, such as partial functions, observability features, etc.
- The main results (e.g. initiality results) will be established in Section 5.

- Exception signatures and exception algebras will be introduced in Section 6 as a particular case of label algebras, and the difference between *exception* and *error* is rigorously defined.
- Exception specifications and their semantics will be defined in Section 7, and they are related to the semantics of label algebras (via a simple translation). Exception algebras allow us to handle all desirable exception handling features, such as implicit propagation of exceptions, recovery, bounded data structures, clarity, terseness, etc. without the usual inconsistency problems raised by exception handling.
- Section 8 contains the fundamental results about exception algebras (directly deduced from the properties of label algebras). It is also shown how these results allow us to handle "structured" exception specifications.
- Section 9 provides a collection of simple examples of exception specifications. They illustrate many powerful aspects of the semantics of exception specifications. We also show that all the mentioned class of exceptional cases ("intrinsic" exceptions, "dynamic" exceptions, bounded data structures) can easily be specified.
- Recapitulation and perspectives can be found in Section 10.

We assume that the reader is familiar with algebraic specifications ([GTW78], [EM85], [GB84]) and with the elementary definitions of category theory [McL71].

2 Crucial aspects of exception handling

We mentioned in the introduction that a good exception handling framework for algebraic specifications must first and foremost be able to cover all kinds of exceptional cases. In this section, we will illustrate two additional crucial aspects that are often neglected, namely clarity and terseness, and we will study their general implications for formal specifications. We will also refer to the other classical desirable aspects.

2.1 Exception handling and programming languages

Let us consider a simple example of algorithm: a function which searches an element `e` in a list. Naive programmers often make the following mistake:

```
current := first ;
while ((current <> nil) and (current.value <> e)) do...
```

This only works if they are lucky with respect to the compiler ![1] Less naive programmers write:

```
current := first ; found := false ;
while ((current <> nil) and (not found))
                                        do found := (current.value = e) ...
```

[1] more precisely if "`and`" is a lazy operator which evaluates first the left hand side argument.

Similar solutions are not acceptable for specifications, because a specification must be *abstract* and *clear*. Moreover the test `current <> nil` is done many times, while the end of the list is exceptional. Experienced programmers add a fictitious `last` cell at the end of a list (thus the empty list contains one cell); they write

```
last.value := e ; current := first ;
while (current.value <> e) do current := current.next ;
  ... ;
```

and the search fails if and only if `current=last` at the end. Of course, this solution is not abstract at all and the solution (as you may have guessed from the beginning of our story) is exception handling. The exception handler plays a role similar to the fictitious cell:

`when Illegal-pointer-access return` ... *(the search has failed).*

The main algorithm is as simple as possible:

```
current := first ;
while (current.value <> e) do current := current.next ;
   ... ;
```

and the search for instance returns the place of `e` if the handler has not been called.

Conclusion: *exception* handling is not only used for *error* handling; it is also a great tool for *clarity* and *terseness*.

- Clarity: the "rare cases" are extracted from the main text (algorithm or specification) so that it becomes easily readable. Notice, by the way, that the notion of "exception" depends on what is considered as rare (one could consider that finding the element is the rare case, and raise an exception when `current.value = e`).
- Terseness: the exception handler, as well as the main text, goes straight to the point. Each statement does not have to deal with the cases that it does not directly concern: this partition of the application domains is handled implicitly by the underlying semantics.

2.2 Exception handling and formal specifications

For formal specifications, clarity and terseness are *a fortiori* an important aim of exception handling. We believe that a formal framework only capable of treating *error* handling is not sufficient; specification and abstraction require *exception* handling. An exception is not necessarily an error; it simply requires a special treatment which has to be separated from the main properties. Thus, errors are simply a particular case of exception.

Clarity should have an important impact on the syntax of formal specifications. It is necessary to distinctly separate the properties reflecting the exceptional cases from the ones of the "normal behaviour." When this partition is not available, it is often necessary to write complex axioms where additional symbols appear (e.g. predicates) to restrict

the scope of the axioms to normal (resp. exceptional) cases. Thus, the text of a formal specification must be partitioned in such a way that the rare cases can be specified as "exceptions" apart from the other properties; the semantics must *implicitly* restrict the scope of the axioms.

Terseness has necessarily an important impact on the semantics of formal specifications; the specialized semantics for each part of the syntax (exceptional/normal properties) must be powerful enough to handle obvious general properties of exceptions. For instance, it is clear that errors should propagate by default (if a is erroneous, then $f(a)$ is also erroneous, except if it is recovered); such properties should not have to be explicitly specified.

Moreover, the following principles have been widely recognized to be crucial for abstract specifications with exception handling ([Gog78a], [GDLE84], [Bid84], [Ber86], [BBC86], [Sch91]):

- each exceptional (or erroneous) case should be declared with some exception name (or error message) which provides enough informations to treat it easily;
- all the relevant properties of exceptional state behaviours should be formally specified;
- an implicit exception propagation rule by default should be provided;
- however various recoveries of the exceptional cases must be possible, related to their exception names.

3 Algebraic specifications with exception handling

The main difficulty of exception handling for algebraic specifications is that all the "simple" semantics that we can imagine lead to inconsistencies. To illustrate this fact, let us try to specify natural numbers with exception handling. We will start with the simple "intrinsic" exception $pred(0)$, and we will progressively add more and more sophisticated exceptional cases. Step by step, we will show that more and more sophisticated semantics are needed. At the end, we show that a clear and terse specification of bounded natural numbers, with certain recoveries, requires semantics based on terms instead of values. Indeed, the specification of bounded natural numbers raises all the main difficulties of exception handling for algebraic specifications.

3.1 Errors as constant operations

A simple idea would be to use the classical ADJ semantics [GTW78], adding a new constant *error* of sort *Nat* and the axiom:

$$pred(0) = error$$

Of course, we have to face error propagation: what is the value of $succ(error)$? A natural idea is to add, for each operation f of the signature, axioms of the form:

$$f(\ldots error \ldots) = error$$

unfortunately, the specification also contains the axiom

$$(1) \qquad x \times 0 = 0$$

thus we get $error = 0$ (with $f = \times$, via the assignment $x = error$). Indeed we meet here the principle that "normal cases" should be distinguished from exceptional cases. The semantics of "normal axioms" should be *implicitly* of the form:

$$x \neq error \implies x \times 0 = 0$$

Notice that the existence of an initial algebra is not ensured in general, as a negative atom appears in the axiom [WB80]. Indeed this fact has already been shown in [GTW78], where an explicit introduction of an OK predicate is proposed.

3.2 Errors and OK predicates

Assuming that the specification also contains a boolean sort *Bool*, we can define an OK_{Nat} predicate which checks if a value is a normal value:

$$\begin{gathered} OK_{Nat}(error) = false \\ OK_{Nat}(0) = true \\ OK_{Nat}(succ(n)) = OK_{Nat}(n) \\ OK_{Nat}(pred(0)) = false \\ OK_{Nat}(pred(succ(n))) = OK_{Nat}(n) \\ \ldots \\ OK_{Nat}(x \times y) = OK_{Nat}(x) \text{ and } OK_{Nat}(y) \end{gathered}$$

If we want to express that an instance of the axiom (1) must be considered only if the two members of the axiom (1) are first checked as normal values, then we write:

$$OK_{Nat}(x \times 0) = true \wedge OK_{Nat}(0) = true \implies x \times 0 = 0$$

and the existence of an initial algebra is ensured. Unfortunately, this approach does not succeed with respect to clarity and terseness, as already pointed out by the authors in [GTW78]: *"the resulting total specification (...) is unbelievably complicated."* It is also shown that the axioms defining OK_{Nat} cannot be automatically generated without inconsistencies ($true = false$); this is particularly obvious when recoveries are allowed. To be convinced, it is sufficient to try to define OK_{Nat} consistently when $succ(pred(0))$ is recovered... (See also [Gog78a] and [Pla82].)

3.3 Errors and partial functions

Clearly, these difficulties result from the explicit introduction of an *erroneous value* in the signature. Moreover, the specification of the OK predicate resembles the specification of definition domains. Thus, a simple idea could be to consider partial functions instead of total functions (e.g. $pred(0)$ being undefined), see for instance the pioneering work of [BW82] (but many other references are relevant too). Unfortunately, specifying exceptions via partial functions is not powerful enough for a full exception handling. For instance exceptional cases can give rise to ulterior recoveries, especially for robust softwares: even if $f(x)$ is not defined, we can require for $g(f(x))$ to be defined (e.g. $succ(pred(0))$). More generally, we must often be able to specify properties concerning exceptional cases, even if they are not recovered. Consequently, exceptional cases must always keep some "semantical meaning," as we must allow specific treatments of exceptional or erroneous values themselves.

Nevertheless, if the specifier is not interested in recoveries and does not want to attach error messages to erroneous values, (s)he probably should use specification semantics based on partial functions.

3.4 Error handling and subsorting

Another strong limitation of the original ADJ approach is that there is only one error value by sort (it can be extented to a finite number, but we often need an infinite number of erroneous values, see Section 3.7). A possible idea of solving this problem is to use subsorting; the OK-part of the sort *Nat* being a subsort *OkNat* of *Nat*. Since the work of Goguen in [Gog78b], the framework of order-sorted algebras has been widely advocated to be a solution for exception handling (see also [FGJM85][GM89]).

For example, it is easy to declare that the sort *OkNat* is generated by 0 and *succ*, that *ErrNat* is the sort reduced to the singleton $\{error\}$, and that *Nat* is the union of *OkNat* and *ErrNat*. Then, we can restrict the scope of the axiom

$$x \times 0 = 0$$

to the sort *OkNat* and this prevent from the inconsistency described above.

Notice that type inference is required in order to determine the scope of an axiom. To be able to deduce that $pred(0)$ belongs to *ErrNat*, and that $pred(x)$ belongs to *OkNat* when x is a positive natural number, a sort *PosNat* is declared, which is equal to $succ(OkNat)$ (it is not difficult to prove that *PosNat* is a subsort of *OkNat*). Then, roughly speaking, the arity of *pred* is specified via overloading:

$$\begin{aligned} pred: \; PosNat &\rightarrow OkNat \\ pred: \; \{0\} &\rightarrow ErrNat \\ pred: \; ErrNat &\rightarrow ErrNat \end{aligned}$$

which implies, for instance, that *pred* can be shown as an operation from *OkNat* to *Nat*. Similarly, the Euclidean division can be specified with the arity

$$div: \ OkNat \ \ PosNat \ \rightarrow \ OkNat$$
$$div: \ Nat \ \ \{0\} \ \rightarrow \ ErrNat$$
$$div: \ Nat \ \ ErrNat \ \rightarrow \ ErrNat$$

and so on.

Unfortunately, things are not this easy. Let us specify the subtraction. The definition domain of "$-$" is the set of all $(a, b) \in OkNat$ such that $a \geq b$. This definition domain cannot be expressed as a Cartesian product of *Nat* subsorts. The solution is to define a new sort *Nat2* which is the Cartesian product $Nat \times Nat$ and to *explicitly* define the domain of "$-$" as a subsort $Dsub$ of *Nat2*. Even if we forget the number of coercions required to type a simple expression (such as $(a - b) - pred(c)$), it remains that the explicit specification of $Dsub$ will not be *terse*:

$$a \in OkNat \implies (a, 0) \in Dsub$$
$$(a, b) \in Dsub \implies (succ(a), succ(b)) \in Dsub$$

The point is that these two typing axioms must be compatible with the semantics of the subtraction, they require an effort from the specifier which is almost as difficult as the definition of the OK predicate of [GTW78]. Moreover, it is somewhat redundant with the axioms defining the subtraction (at least if they are well chosen):

$$a - 0 = a$$
$$succ(a) - succ(b) = a - b$$

Indeed, we have in mind that it is sufficient to specify the subsort $OkNat$. Roughly speaking, if the axioms defining "$-$" allow us to find a result for $(a - b)$ in $OkNat$ then (a, b) belongs to $Dsub$, or else $(a - b)$ is exceptional. The same remark applies to *pred*: the explicit specification of *PosNat* is unwanted.

Notice moreover that the propagation of errors is actually *not* implicit, since the definition domain of each operation should be explicitly defined on all the elements of a sort.

The main advantage of the approaches based on subsorting is that the specification style fulfills the clarity criterion in general. Moreover, the names of the (erroneous) subsorts can be used to reflect exception names (or error messages) in such a way that a precise error handling can be performed. The lack of terseness is the main disadvantage of these approaches (see also Section 3.6 where another strong limitation of subsorting is explained).

More precisely, the terseness criterion for exception handling is better fulfilled when the semantics are based on a declaration of the "*Ok-codomain*" of the operations rather than their "*Ok-domain*." The reason is simple: in general, all the operations of a data type share the same Ok-codomain, while each of them has its own Ok-domain. The first framework that took advantage of this idea (even if it was not explicitly analysed this way) is the one of [GDLE84] where the Ok-part of a sort is described via "safe" operations.

3.5 Safe and unsafe operations

The simplest idea to describe the Ok-part of each sort is to distinguish a set of operations (subset of the signature) that generates the Ok-values. In [GDLE84][Gog87], the signature Σ is partitioned into "safe" and "unsafe" operations. For example, 0, *succ* and $+$ are safe operations because when they are applied to Ok-arguments, they always return Ok-results; on the contrary, *pred* and $-$ are unsafe because 0 is Ok but $pred(0)$ is erroneous, and, for instance, 0 and $succ(0)$ are Ok but $0 - succ(0)$ is erroneous. The main advantage of this approach is that such a simple syntactic classification of functions is sufficient to describe the Ok and erroneous part of each sort; the ok-values are those generated by the safe operations, all the other values are automatically erroneous. For example, the axioms defining the subtraction are sufficient to automatically deduce its "Ok-domain;"

$$n - 0 = n$$
$$n - succ(m) = pred(n - m)$$

it is not difficult to prove that $(a - b)$ has an Ok-value (i.e. is in the equivalence class of a term generated by 0, *succ* and $+$) if and only if a is greater or equal to b.

As shown in [GDLE84], this idea is not fully sufficient to solve the inconsistencies mentioned so far. Let us return to the axiom

$$x \times 0 = 0$$

and let us consider an instance of x which is an erroneous value (say *error*). We would still have that $error \times 0 = 0$. This does not induce an inconsistency because $error \times 0$ is not necessarily equal to *error* in the approach of [GDLE84][Gog87]. More precisely, since the axioms allow us to find an Ok-value (0) to $(error \times 0)$, it is automatically recovered (according to the "codomain driven" strategy). Of course, this implicit recovery is not necessarily wished by the specifier, and we must provide a way of preventing it if necessary. This is the reason why the authors introduce a special type of variables (often denoted as "x_+") which can only serve for Ok-values. Then, the previous implicit recovery can be avoided by writing

$$x_+ \times 0 = 0$$

where the assignment $[x_+ \leftarrow error]$ is not allowed. (This special kind of variables is also used in [Bid84], but the proposed semantics is more complicated and gives less usable results, in particular because the initial algebra does not exist).

One of the main advantages of the framework of [GDLE84][Gog87] is that, given a set of positive conditional axioms, a least congruence exists. Consequently an initial algebra exists, a left adjoint functor to the forgetful functor exists, and parameterization can be easily defined. Thus, structured specifications with error handling features can be easily studied in this framework.

Moreover the terseness criteria is satisfied, in particular because the erroneous cases have not to be explicitly characterized. Clarity is also better achieved than with all the approaches mentioned above. However, in practice, the specifier has to be very careful in deciding when a "normal variable" (x) or an "Ok-variable" (x_+) should be used in an

axiom. This is indeed due to the fact that this approach does not offer a semantical distinction between "normal axioms" and "exceptional axioms" (see Section 2.2 above). Clarity would be improved if such a distinction were provided.

An extension of this approach to order sorted algebras exists [Gog83][Gog84]. All the mentioned advantages remain, while preserving the simplicity of the semantics.

As already pointed out in [Ber86][BBC86], the main problem of this framework is that bounded data structures cannot be specified. The reason is simple; for bounded data structures almost all the operations are unsafe, except a few constants. For example, *succ* and + are not safe for bounded natural numbers ($succ(Maxint)$ is erroneous while *Maxint* is Ok); consequently the Ok-part of the sort *Nat* would be reduced to 0 (at least in the initial algebra).

3.6 Bounded data structures and recovery axioms

The approaches mentioned above give solutions to the algebraic treatment of "intrinsic errors" (such as $pred(0)$), with implicit error propagation and possible recovery, but they are not able to treat the other kind of errors mentioned in Section 1, especially bounded data structures. Nevertheless, software engineering requires a careful treatment of bounded data structures. If bounded data structures are not taken into account at the specification level, then almost all the specified properties are actually false; and precisely, in practice, softwares require a strong verification and validation effort near the bounds of the underlying data structures.

Let us sketch out a simple example to give an idea of the difficulties raised by bounded data structures for algebraic specifications, especially when recoveries are allowed. To specify bounded natural numbers it is indeed not too difficult to specify that all the values belonging to $[0 \dots Maxint]$ are Ok-values [BBC86]; let us assume that this is done. We also have to specify that the operation *succ* raises an exception when applied to *Maxint*, e.g. *TooLarge*; let us assume that this is done too. When specifying the operation *pred*, we must have the following axiom:

$$pred(succ(x)) = x \tag{2}$$

which is a "normal property" and, as such, should be understood with certain implicit preconditions such as "*if x and $succ(x)$ are Ok-values*" for example. Let us assume now that we want to recover all *TooLarge* values on *Maxint*. We will then necessarily have $succ(Maxint) = Maxint$.

Since these two values are equal, we will have to choose: either both of them are erroneous values, or both of them are Ok-values. The first case is not acceptable because it does not cope with our intuition of "recovery." (Moreover, when considering the value $m = Maxint - 1$ we clearly need that $pred(Maxint) = m$, as a particular case of our "normal property" about *pred*. Thus $succ(m) = Maxint$ must be considered as a normal value.) Unfortunately, since $succ(Maxint)$ is then a normal value, $x = Maxint$ is an acceptable assignment for our "normal property" and we get the following inconsistency:

$$m = pred(Maxint) = pred(succ(Maxint)) = Maxint$$

which propagates in the same way, and all values are equal to 0.

Remark 1 : A possible reaction to this inconsistency could be to say that *"the specifier should not have written such an inconsistent axiom; he should have been careful and written something like*

$$x \le m = true \implies pred(succ(x)) = x$$
$$pred(Maxint) = m$$

because (s)he knew that $succ(Maxint) = Maxint$*."* Our claim is that this way of thinking contradicts the terseness and clarity principles explained in Section 2. Exception handling must allow the specifier to say *"I declared* $succ(Maxint)$ *exceptional, consequently I should not have to worry about it when I write a normal property; the semantics must discard implicitly the assignment* $succ(Maxint)$ *from the set of acceptable assignments."*

Indeed, it is precisely the difference that we make between "exception handling" and "error handling." The term $succ(Maxint)$ is not erroneous but it is exceptional; the semantics must take this fact into account.

This leads to the following idea: the term $Maxint$ is an acceptable assignment for the variable x in the equation (2) while $succ(Maxint)$ is not, even though $Maxint$ and $succ(Maxint)$ have the same value. The term $Maxint$ is not exceptional while the term $succ(Maxint)$ is exceptional. Thus, exception handling requires taking care of terms inside the algebras and good functional semantics for exception handling should allow such distinctions. This idea has been formalized in [Ber86][BBC86], where "Ok-terms" are declared instead of the "safe operations" of [GDLE84]. In this framework, the term $succ^{Maxint}(0)$ is "labelled" by Ok while the term $succ^{Maxint+1}(0)$ is not;[2] and the acceptable assignments of a normal property (called "Ok-axiom") are implicitly restricted to Ok-terms only. This approach solves the inconsistencies generated by the recovery $succ(Maxint) = Maxint$. The declaration of Ok-terms looks like

$$succ^{Maxint}(0) \;\epsilon\; Ok$$
$$succ(n) \;\epsilon\; Ok \;\;\Rightarrow\;\; n \;\epsilon\; Ok$$

Let us point out that subsorting (see Section 3.4 above) cannot be used to specify such bounded data structures with recoveries. The axiom (2) necessarily gives rise to a similar paradox because sorts are attached to values. Two terms having the same value must share the same subsorts; consequently $Maxint$ and $succ(Maxint)$ cannot be distinguished.

Another crucial idea of [BBC86] is that several exceptional cases can require the same kind of treatment, while keeping distinct values; they are grouped under common exception names. In this framework, exception names are reflected by *labels*, and the exceptional values are *labelled* by their corresponding exception name. We explain in the next subsection the reasons why the introduction of labels is suitable. From this point of view, we follow the main ideas of [BBC86] but our syntax is considerably simpler, as well as the semantics (moreover we give an example below that cannot be treated by [BBC86]).

[2] $succ^i(0)$ is an abbreviation for $succ(succ(\ldots(0)\,))$ where $succ$ appears i times.

3.7 About values, terms and labels

Let us investigate what "exception names" could be in the framework of algebraic specifications. A simple approach is to consider exception names as additional constants of the signature (generalizing the single *error* constant of [GTW78] described in Section 3.1). For example we could add the constant *TooLarge*, of sort *Nat*, and specify that every natural number greater than *Maxint* is equal to *TooLarge*. This approach does not allow a powerful exception handling in general. For example if we consider the expression $((m+n)-p)$ then it is erroneous when $(m+n)$ is *TooLarge*, even if the final result would be less than *Maxint* (by error propagation; $(m+n)$ being equal to *TooLarge*). Of course the specifier may want to allow a rearrangement of the algebraic expression in order to recover its result. A good exception handling must allow to specify such recoveries, for instance:

$$m+n = TooLarge \ \wedge \ p \leq m = true \quad \Longrightarrow \quad (m+n)-p = (m-p)+n$$

Obviously, *TooLarge* being a constant, we get inconsistencies, for example, when $(m+n) = TooLarge$ and $p \leq m$:

$$(m-p)+n = (m+n)-p = TooLarge - p = (m+succ(n))-p = (m-p)+succ(n)$$

Indeed, exceptions having the same name should clearly get the same *exceptional treatment*, not necessarily the same *value*. Consequently, exception names are not values.

Another simple idea is to consider exception names as (sub)types, as in the order sorted approach described in Section 3.4. A similar idea is to consider exception names as (unary) predicates on values; it is more or less the approach followed by [BBC86] where the "exception labels" are carried by values. We have shown in Section 3.6 above that the special label "Ok" cannot be carried by values; it must be carried by terms. The following example shows that exception names must also be carried by terms, not values.

Example 2 : Let us assume that every value of the form $succ^i(Maxint)$ $(i \geq 1)$ is labelled by *TooLarge* (or belongs to the subsort *TooLarge* of *Nat*). Let us assume that we want to recover every *TooLarge*-value on *Maxint*. A possible way of expressing this recovery is to say *"if the operation succ raises the exception TooLarge, then do not perform it."* It is formally specified as:

(3) $$succ(n) \ \epsilon \ TooLarge \quad \Rightarrow \quad succ(n) = n$$

When the exception name *TooLarge* is carried by *values*, the term $succ(Maxint)$ being equal to the term *Maxint*, both of them are labelled by *TooLarge*. Let $m = Maxint - 1$, we get the following inconsistency:

$$Maxint = succ(m) = m$$

because $Maxint = succ(m)$ is labelled by *TooLarge*, thus axiom (3) applies. This inconsistency propagates in the same way, and all values are equal to 0.

Notice that subsorting [FGJM85] gives rise to a similar paradox because sorts are attached to values. Two terms having the same value must share the same subsorts;

thus $succ(Maxint)$ and $Maxint$ cannot be distinguished, and this example would lead to the same inconsistency.

Consequently, in the frameworks of [FGJM85] and [BBC86], it was not possible to specify this kind of recovery. This was indeed the case for all existing algebraic framework for exception handling, because exception names (if provided) were always carried by values.

Nevertheless, the solution is simple: even if $succ(Maxint)$ is recovered on $Maxint$, the exception name $TooLarge$ must not be propagated to $Maxint$. Exception names must not go through recoveries. As a consequence, exception names must be treated in a similar way as the label Ok; they must be carried by terms, not by values. Roughly speaking, exception handling requires a special "typing" of *terms*. We shall call *labels* these special "types," reflecting exception names. Since labels are neither additional constants nor additional sorts of the signature, they form a third component of the signature. From this point of view, the label algebras defined below are an extension of more standard algebraic approaches with "multityping" such as order sorted algebras [Gog78b][FGJM85].

All these considerations have been our main motivation to develop the framework of *label algebras*. The rest of this paper is devoted to rigorously defining and studying label specifications, label algebras and their applications.

Usually, algebras are (heterogeneous) sets of values [GTW78][EM85]. Let us remember that a signature is usually a couple $< S, \Sigma >$ where S a finite set of sorts (or type names) and Σ is a finite set of operation names with arity in S; the objects (algebras) of the category $Alg(\Sigma)$ are heterogeneous sets, A, partitioned as $A = \{A_s\}_{s \in S}$, and with, for each operation name "$f : s_1 \ldots s_n \to s$" in Σ ($0 \leq n$), a total function $f_A : A_{s_1} \times \ldots \times A_{s_n} \to A_s$; the morphisms of $Alg(\Sigma)$ (Σ-morphisms) being obviously the sort preserving, operation preserving applications.

As a consequence of our approach, (labelling of) terms must also be considered as "first class citizen objects." Given an algebra A, the satisfaction of a normal property must be defined using terms (instead of the usual definition which only involves values). A simple idea could be to consider both A and T_Σ (the ground term algebra over the signature Σ) when defining the satisfaction relation. Unfortunately, such an approach does not allow satisfactory treatments of the non finitely generated algebras, i.e. algebras such that the initial Σ-morphism from T_Σ to A is not surjective. In a general way, finitely generated algebras are not powerful enough to cope with enrichment, parametrization or abstract implementations. Moreover, let us consider for instance some algebra A containing a value α which is not reachable by a ground term, α and $succ(\alpha)$ being supposed non exceptional. If A satisfies the "normal property" $pred(succ(x)) = x$ then any reader of the specification has the right to assume that $pred(succ(\alpha)) = \alpha$. Consequently, the satisfaction of a "normal property" must also consider unreachable values. How is one to deal with both terms and non reachable values ? Indeed a similar question has already been solved for some time, e.g. in [GTW78], when defining the free synthesis functor F: given any algebra A, $F(A)$ is defined as the quotient of $T_\Sigma(A)$ by some well chosen congruence. The algebra $T_\Sigma(A)$ being crucial in our approach, let us remember its definition.

- Given a heterogeneous "set of variables" $V = \{V_s\}_{s \in S}$, the *free Σ-term algebra with*

variables in V is the least Σ-algebra $T_\Sigma(V)$ (with respect to the preorder induced by the Σ-morphisms) such that $V \subset T_\Sigma(V)$.

- Since V is not necessarily finite or enumerable, we can consider in particular $T_\Sigma(A)$ for every algebra A. An element of $T_\Sigma(A)$ is a Σ-term such that each leaf can contain either a constant of the signature, or a value of A.

- For example, if $A = \mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$ is the algebra of all integers over the signature *{zero,succ,pred}*, then $succ(succ(zero))$, $succ(succ(0))$, $succ(1)$, etc. are *distinct* elements of $T_\Sigma(\mathbb{Z})$, even though they have the same value when evaluated in $\mathbb{Z}$.

The main technical point underlying our framework is to systematically use $T_\Sigma(A)$ directly inside the *label algebras* in order to have a refined treatment of labelling. For example, considering the non finitely generated algebra mentioned above, α, $succ(\alpha)$, $pred(succ(\alpha))$, as well as $Maxint$ and $succ(Maxint)$, are distinct elements of $T_\Sigma(A)$ ($succ(Maxint)$ being the only exceptional term, labelled by $TooLarge$). This allows us to have a very precise definition of the satisfaction relation, using assignments with range in $T_\Sigma(A)$ instead of A.

Intuitively, a term reflects the "history" of a value; it is a "sequence of calculi" which results in a value. Of course, several histories can provide the same value. This is the reason why labelling is more powerful than typing: it allows to "diagnose" the history in order to apply a specific treatment or not. Nevertheless, we must be able to relate each term to its final value. This is easily obtained via the canonical evaluation morphism:

$$eval_A : T_\Sigma(A) \longrightarrow A$$

deduced from the Σ-algebra structure of A:

$$\forall a \in A,\ eval_A(a) = a \quad \text{(remember that } A \subset T_\Sigma(A))$$
$$\forall f \in \Sigma,\ \forall t_1 \ldots t_n \in T_\Sigma(A),\ eval_A(f(t_1 \ldots t_n)) = f_A(eval_A(t_1) \ldots eval_A(t_n))$$

Of course, *in the end*, the satisfaction of an equality must be checked on values; thus, $eval_A$ is a crucial tool for defining the satisfaction relation on equational atoms. However, the considered assignments can be precisely restricted to certain kinds of terms/histories *before* checking equalities on values, and this is the reason why all the inconsistencies mentioned before can be solved via label algebras.

We shall use the following simplified notations:

Notation 3 : Given a Σ-algebra A, $T_\Sigma(A)$ will be denoted by $\overline{A}$. Moreover, let $\mu : A \to B$ be a Σ-morphism, $\overline{\mu} : \overline{A} \to \overline{B}$ denotes the canonical Σ-morphism which extends μ to the corresponding free algebras. Let us note that: $\mu \circ eval_A = eval_B \circ \overline{\mu}$.

4 Label algebras

4.1 Basic definitions

Definition 4 : A *label signature* is a triple $\Sigma L = < S, \Sigma, L >$ where $< S, \Sigma >$ is a (usual) signature and L is a (finite) set of *labels*.

Definition 5 : Given a label signature $\Sigma L = < S, \Sigma, L >$, a ΣL-algebra is a couple $\mathcal{A} = (A, \{l_A\}_{l \in L})$ where:

- A is a Σ-algebra,
- $\{l_A\}_{l \in L}$ is a L-indexed family such that, for each l in L, l_A is a subset of $\overline{A}$.

Notice that there are no conditions about the subsets l_A: they can intersect several sorts, they are not necessarily disjoint and their union $(\bigcup_{l \in L} l_A)$ does not necessarily cover $\overline{A}$.

Definition 6 : Let $\mathcal{A} = (A, \{l_A\}_{l \in L})$ and $\mathcal{B} = (B, \{l_B\}_{l \in L})$ be two ΣL-algebras, a ΣL-morphism $h : \mathcal{A} \to \mathcal{B}$ is a Σ-morphism from A to B such that $\overline{h} : \overline{A} \to \overline{B}$ preserves the labels: $\forall l \in L,\ \overline{h}(l_A) \subset l_B$.

When there is no ambiguity about the signature under consideration, ΣL-algebras and ΣL-morphisms will be called *label algebras* and *label morphisms*, or even algebras and morphisms. Given a label signature ΣL, label algebras and label morphisms (with the usual composition) clearly form a category:

Definition 7 : The category of all ΣL-algebras is denoted by $Alg_{Lbl}(\Sigma L)$.

If there exists a label morphism from $\mathcal{A}$ to $\mathcal{B}$, we also note $\mathcal{A} \leq \mathcal{B}$ (the preorder induced by the morphisms).

Notations 8 : The ΣL-algebra $\mathcal{T}_{\Sigma L}$ is defined by:

- the underlying Σ-algebra of $\mathcal{T}_{\Sigma L}$ is the ground terms algebra T_Σ
- for each l in L, $l_{\mathcal{T}_{\Sigma L}}$ is empty.

The ΣL-algebra $\mathcal{T}riv$ is defined by:

- the underlying Σ-algebra of $\mathcal{T}riv$ is the trivial algebra $Triv$ which contains only one element in $Triv_s$ for each s in S.
- for each l in L, $l_{\mathcal{T}riv} = \overline{Triv}$.

The ΣL-algebra $\mathcal{T}_{\Sigma L}$ (resp. $\mathcal{T}riv$) is clearly initial (resp. terminal) in $Alg_{Lbl}(\Sigma L)$. Moreover, as usual, a ΣL-algebra $\mathcal{A}$ is called *finitely generated* if and only if the initial ΣL-morphism from $\mathcal{T}_{\Sigma L}$ to $\mathcal{A}$ is an epimorphism. It is clear from the definitions that $\mathcal{A}$ is finitely generated if and only if the underlying morphism from T_Σ to A is surjective, which means that every value of A is reachable by a ground term.

Definitions 9 : The full subcategory of $Alg_{Lbl}(\Sigma L)$ containing all the finitely generated algebras is denoted by $Gen_{Lbl}(\Sigma L)$. Moreover, the signature ΣL is said *sensible* if and only if $\mathcal{T}riv$ belongs to $Gen_{Lbl}(\Sigma L)$.

The category $Gen_{Lbl}(\Sigma L)$ has the same initial object as $Alg_{Lbl}(\Sigma L)$, and if ΣL is sensible (i.e. if there exists at least one ground term of each sort) then it has the same terminal object too.

Not surprisingly, a "label specification" will be defined by a (label) signature and a set of well formed formulae (axioms):

Definition 10 : Given a label signature ΣL, a ΣL-*sentence* (or *axiom*) is a well formed formula built on

- *equational atoms* of the form $(u = v)$ where u and v are Σ-terms with variables, u and v belonging to the same sort,
- *labelling atoms* of the form $(w \epsilon l)$ where w is a Σ-term with variables and l is a label belonging to L,
- *connectives* belonging to $\{\neg, \wedge, \vee, \Rightarrow\}$.

(Every variable is implicitly universally quantified.)[3]

The predicate " ϵ " should be read "*is labelled by*".

Definition 11 : A *label specification* (or *presentation*) is a pair $SP = <\Sigma L, Ax>$ where ΣL is a label signature and Ax is a set of ΣL-sentences.

The *satisfaction relation* is indeed the crucial definition of this section. It is of first importance to remark that we consider assignments with range in $\overline{A} = T_\Sigma(A)$ (terms) instead of A (values):

Definition 12 : Let $\mathcal{A} = (A, \{l_A\}_{l \in L})$ be a ΣL-algebra.

- $\mathcal{A}$ satisfies $(u = v)$, where u and v are two terms of the same sort in $\overline{A}$, means that, in A, $eval_A(u) = eval_A(v)$ [$eval_A$ being the canonical evaluation morphism from $\overline{A}$ to A and the symbol "=" being the set-theoretic equality in the carrier of A].
- $\mathcal{A}$ satisfies $(w \epsilon l)$, where $w \in \overline{A}$ and $l \in L$, means that $w \in l_A$ [the symbol "$\in$" being the set-theoretic membership].
- Given a ΣL-sentence φ, $\mathcal{A}$ *satisfies* φ, denoted by $\mathcal{A} \models \varphi$, means that for all assignments $\sigma : V \to \overline{A}$ (V covering all the variables of φ), $\mathcal{A}$ satisfies $\sigma(\varphi)$ according to the "ground atomic satisfaction" defined above and the truth tables of the connectives.

[3] Allowing existential quantifiers is not difficult at all, but the management of the assignments for the satisfaction relation becomes rather tedious. . . and this extension is not required for defining exception algebras.

A label algebra satisfies a label specification if and only if it satisfies all its axioms.

Given a label specification SP, the full subcategory of $Alg_{Lbl}(\Sigma L)$ containing all the algebras satisfying SP is denoted by $Alg_{Lbl}(SP)$. (A similar notation holds for Gen_{Lbl}.)

Notice that $Alg_{Lbl}(SP)$ or $Gen_{Lbl}(SP)$ can be empty categories (for example when SP contains φ and $\neg\varphi$). Providing that the axioms of SP never contain the connective "$\neg$", $Alg_{Lbl}(SP)$ has the same terminal object as $Alg_{Lbl}(\Sigma L)$: $\mathcal{T}riv$. However, as usual, initiality results can be easily obtained only for positive conditional specifications [WB80]. These results are provided in Section 5.

Definition 13 : A ΣL-sentence is called *positive conditional* if and only if it is of the form:

$$a_1 \wedge \ldots \wedge a_n \;\Rightarrow\; a$$

where the a_i and a are (positive) atoms (if $n = 0$ then the sentence is reduced to a). A specification is called *positive conditional* if and only if all its axioms are positive conditional sentences.

4.2 Some applications of label algebras

Although we have introduced the theory of label algebras as a general frame for exception handling purpose, it can also be used for many other purposes. We have mentioned so far that labels can be used to represent exception names. Indeed, labels provide a great tool to express several other features already developed in the field of (first order) algebraic specifications. In this section, we outline some possible applications of the framework of label algebras.

We have mentioned in Section 3.7 that the framework of label algebras can be shown as an extension of more standard algebraic approaches based on "multityping." More precisely, we can *specify multityping* by means of label specifications. Indeed the difference between a label and a type is that labels are carried by terms (in $\overline{A}$) while type names are carried by values (in A). Thus a label l can easily play the role of a type name: it is sufficient to saturate each fiber of $eval_A : \overline{A} \to A$ which contains a term labelled by l. This is easily specified by a ΣL-sentence of the form:

$$x \,\epsilon\, l \;\wedge\; x = y \;\Longrightarrow\; y \,\epsilon\, l$$

where x and y are variables. For every model A satisfying such axioms for l belonging to L, two terms u and v of $\overline{A}$ having equal values in A are necessarily labelled by the same labels, thus labels can play the role of types. Notice that we should write one axiom of this form for each sort belonging to S because the variables x and y are typed with respect to S in our framework. Nevertheless, as far as we intend to simulate types by labels, S should be a singleton. Thus, the "typing" of terms, as well as variables, becomes explicit in the precondition of each axiom. Therefore, this approach leads to consider typing as "membership constraints." For finitely generated algebras, such a specification style facilitates theorem proving, as demonstrated in [Smo86][Com90].

An advantage of such an approach is that additional properties about types, according to the needs of the considered application, can be easily specified within the same framework. For example, let us consider a property such as $s \leq s'$ between two sorts in the framework of *order sorted algebras* [FGJM85]. It can be specified within the framework of label specifications:

$$x \mathrel{\epsilon} s \implies x \mathrel{\epsilon} s'$$

where s and s' are labels which simulate the corresponding (sub)sorts.

Algebraic specifications with *partial functions* can also be reflected via label specifications. Algebraic specifications for partial algebras often rely on an additional predicate D which is used to specify the definition domain of each operation of the signature ([BW82] and others). Thus, atoms are either equalities, or of the form $D(t)$, where t is a term with variables. It is of course not difficult to translate $D(t)$ to $(t \mathrel{\epsilon} IsDefined)$; we simply have to specify the propagation of the definition domains with respect to any operation f of the signature:

$$f(x_1, \ldots, x_n) \mathrel{\epsilon} IsDefined \implies x_1 \mathrel{\epsilon} IsDefined \wedge \ldots \wedge x_n \mathrel{\epsilon} IsDefined$$

Then, the label $IsDefined$ can be used in the preconditions of the axioms defining the partial operations in such a way that every label algebra $\mathcal{A}$ satisfying the resulting label specification has the property that $eval_A(IsDefined_A)$ is a subset of A that behaves like a partial algebra satisfying the original specification (see also [AC91]).

In the same way, labels can be used to give a refined semantics of the predefined *predicates of specification languages*. For example in Pluss [Bid89], an expression of the form "t `is defined when` *something*" can be reflected by the following label axiom:

$$\textit{something} \implies t \mathrel{\epsilon} IsDefined$$

More generally, labels are indeed unary predicates on terms; thus, they can be at least used as *predicates* on values (using the label axiom already mentioned for multityping). The advantage of such predicates is that their semantics is not defined via a hidden boolean sort: using booleans to define predicates is often unsatisfactory because it assumes that the specification is consistent with respect to boolean values. In this way, labels can advantageously be used in a specification to provide additional informations about the specified data types, without any exception handling connotation. An example is given in Figure 4.2.

Another possible application of the framework of label algebras is that of algebraic specifications with *observability* issues. A crucial aspect of observational specifications is that "what is observable" must be carefully specified. It is often very difficult to prove that two values are observationally equal (while it is sufficient to exhibit two observations which distinguish them to prove that they are distinct). In [Hen89], R. Hennicker uses a predicate *Obs* to characterize the observable values. This powerful framework leads to legible specifications and it provides some theorem proving methods. Of course, the predicate *Obs* can be reflected by a label. Moreover, it has been shown in [BBK91] and [BB91] that there are some specifications which are inconsistent when observability is carried by values. It is shown that these inconsistencies can be avoided when observability

1— labels as simple predicates

$0 \epsilon Even$
$n \epsilon Even \Rightarrow succ(n) \epsilon Odd$
$n \epsilon Odd \Rightarrow succ(n) \epsilon Even$
$exp(n, 0) = succ(0)$
$succ(m) \epsilon Odd \Rightarrow exp(n, succ(m)) = exp(n, m) \times n$
$m \in Even \Rightarrow exp(n, m) = exp(n \times n\ ,\ m\ div\ succ(succ(0))\ )$

is expressed with respect to a subset ΣObs of the signature Σ (leading consequently to a subset of the *terms* instead of *values*). The framework of [BB91] introduces two distinct notions that reflect a hierarchy in the definition of observability. The terms that only contain operations belonging to ΣObs are said to "allow observability" (the other ones can never be observed). Then, a term "allowing observability" really becomes "observable" only if it belongs to an observable sort. It is not difficult to reflect the observational hierarchy defined in [BB91] by using two distinct labels denoted *AllowsObs* and *Obs*. For each operation f allowing observability (i.e. belonging to the considered subset ΣObs of the signature), it is sufficient to consider the following label axiom:

$$x_1 \epsilon AllowsObs \wedge \ldots \wedge x_n \epsilon AllowsObs \implies f(x_1, \ldots, x_n) \epsilon AllowsObs$$

The fact that a term allowing observability becomes observable if and only if it belongs to an observable sort s can easily be specified by the label axiom (one axiom for each observable sort):

$$x \epsilon AllowsObs \implies x \epsilon Obs$$

where x is a variable of sort s. Hopefully, the advantages of the Hennicker's approach could be preserved, since they mainly rely on the explicit specification of the predicate *Obs*.

Summing up, the framework of label algebras is clearly not directly usable by a "working specifier." All the possible applications mentioned above require some generic label axioms which must be *implicit*. These axioms should be considered as modifiers of the semantics, in order to preserve *clarity* and *terseness* of the specifications. Thus, the framework of label algebras provides us with "low level" algebraic specifications. When an algebraic specification *SP* is written according to some special semantics (e.g. observational specifications or exception algebras), it has to be "compiled" (translated) to a label specification $Tr(SP)$. In some cases, the so-called *partial evaluation constraint* is useful in order to facilitate these translations. This constraint is described in the next subsection.

4.3 The partial evaluation constraint

Almost all the examples given in the previous subsection have the following property: if a term $t \in \overline{A}$ is labelled by l then every partial evaluation of t is still labelled by l.

For instance, let us return to the application of label algebras to observability. Let us consider the algebra $A = \mathbb{Z}$ and let us assume that A is observed via some boolean terms of $\overline{A}$. If the term $[pred(pred(0)) \leq succ(succ(0))]$ is labelled by Obs, then we clearly would like $[pred(-1) \leq succ(1)]$ to also be observable (i.e. labelled by Obs), as well as $[-2 \leq succ(1)]$, $[-2 \leq 2]$, or $true$ itself. Similarly, when exception handling is involved, if the term $((3 + 4) - 5)$ is an Ok-term (i.e. labelled by Ok), then we probably would like for $(7 - 5)$, or 2, to be also labelled by Ok. More generally, the terms labelled by Ok are sequences of calculi which contain only "normal treatments." This means that an exceptional treatment cannot appear at any stage of the evaluation of such terms. Thus, any partial evaluation of any term labelled by Ok can also be labelled by Ok.

On the other hand, this partial evaluation constraint cannot be required for labels which reflect exception names. It would disallow some recovery cases. For example, let us assume that the term $succ(Maxint)$ is exceptional and labelled by $TooLarge$. Nevertheless it can be recovered on $Maxint$, e.g. via the axiom:

$$succ(n) \mathrel{\epsilon} TooLarge \quad \Rightarrow \quad succ(n) = n$$

We have shown in Example 2 that the constant $Maxint$ (which is an evaluation of $succ(Maxint)$) must not be labelled by $TooLarge$.

Intuitively, since a term of $\overline{A}$ reflects the history of a value, if this history does not raise exceptional cases then it can be entirely or partially forgotten (via partial evaluations); but it cannot be forgotten if the term is exceptional because exceptional treatments are specified with respect to this history (often via pattern matching). Summing up, some labels of a label signature should follow the so called partial evaluation constraint while some other ones should not.

Definition 14 : A *constrained label signature* is a triple $\widehat{\Sigma L} = < S, \Sigma, \widehat{L} >$ where $< S, \Sigma >$ is a usual signature and $\widehat{L}$ is a couple (L, C) such that L and C are disjoint sets of labels. The labels of L are called "unconstrained;" the ones of C are called "constrained." We shall note $\widetilde{L} = L \cup C$ the set of all labels, and $\Sigma\widetilde{L} = < S, \Sigma, \widetilde{L} >$ the corresponding (unconstrained) label signature.

A constrained label signature can be seen as a label signature (with respect to $\widetilde{L}$) such that a subset of constrained labels (C) is distinguished. Before defining "constrained label algebras," let us first introduce some terminology about terms.

Definitions 15 : Let A be a Σ-algebra and let t be a term in $\overline{A} = T_\Sigma(A)$.

- Let u be (an occurrence of) a subterm of t and let v be any term in $\overline{A}$ of the same sort as u. The term $t[u \leftarrow v]$ is the term of $\overline{A}$ obtained by replacing (the considered occurrence of) u by v in t.
- When v is the value $eval_A(u)$ of A (remember that A is included in $\overline{A} = T_\Sigma(A)$), the term $t[u \leftarrow eval_A(u)]$ is a *partial evaluation* of t. More generally, a term t' is a partial evaluation of t if it can be obtained via a finite sequence of such partial evaluations.

For example, $(7+2)$ and 9 are partial evaluations of the term $((4+3)+(1+1))$, but $(6+3)$ is not a partial evaluation of this term.

Remarks 16 :

- If t' is a partial evaluation of t then $eval_A(t') = eval_A(t)$. The converse property is false.
- For every term t of $\overline{A}$, the (constant) term $eval_A(t)$ is a partial evaluation of t (obtained via $u = t$).

Definition 17 : Given a constrained label signature $\widehat{\Sigma L}$. A $\widehat{\Sigma L}$-algebra is a label algebra over the signature $\Sigma\widetilde{L}$ which satisfies the following *partial evaluation constraint*: for every label l in C and for every term t in $\overline{A}$, if t belongs to l_A then all partial evaluations of t still belong to l_A.

$Alg_{Lbl}(\widehat{\Sigma L})$ is the full subcategory of $Alg_{Lbl}(\Sigma\widetilde{L})$ which contains all the $\widehat{\Sigma L}$-algebras. A similar notation holds for Gen_{Lbl}.

Intuitively, the stability of labelling by constrained labels with respect to partial evaluation means that, as soon as a term has been labelled, one can forget its "old history" without modifying its constrained labels. Equivalently, since a term represents a sequence of calculi, it means that constrained labelling does not depend on the particular computational strategy. The only point which matters is that there exists at least one strategy which yields the constrained label. (It is not the case for unconstrained labels and, for exception handling purposes, exception labels are never constrained.)

Remarks 18 :

- $\mathcal{T}_{\Sigma\widetilde{L}}$ and $\mathcal{T}riv$ are constrained label algebras, whatever C is.
- For every constrained label algebra, constrained labels are compatible with $eval_A$ in the sense that $eval_A(l_A) \subset l_A$. Notice that the converse property is false: labels which satisfy $eval_A(l_A) \subset l_A$ are not necessarily stable by partial evaluation (see Remarks 16 above).
- The partial evaluation constraint cannot be specified using label axioms over the signature $\Sigma\widetilde{L}$. Let us note that in particular the following attempt is wrong: to consider, for each operation f of the signature, all the axioms of the form

  $$f(x_1, \ldots, x_i, \ldots, x_n) \mathrel{\epsilon} l \;\wedge\; x_i = x \implies f(x_1, \ldots, x, \ldots, x_n) \mathrel{\epsilon} l$$

  It would imply for example that $((2+3)-1)$ and $((6-1)-1)$ have the same label, while none of these two terms is a partial evaluation of the other one.

Definitions 19 :

- Given a constrained signature $\widehat{\Sigma L}$, a $\widehat{\Sigma L}$*-sentence* is simply a $\Sigma\widetilde{L}$-sentence (as defined in Definition 10).

- A *constrained label specification* is a pair $\widehat{SP} = < \widehat{\Sigma L}, Ax >$ where $\widehat{\Sigma L}$ is a constrained label signature and Ax is a set of $\Sigma\widetilde{L}$-sentences.
- The category $Alg_{Lbl}(\widehat{SP})$ is the full subcategory of $Alg_{Lbl}(\widehat{\Sigma L})$ containing all the algebras satisfying Ax (according to the satisfaction relation defined in Definition 12). An object of $Alg_{Lbl}(\widehat{SP})$ is called a $\widehat{SP}$-algebra.

Similar definitions hold for $Gen_{Lbl}(\widehat{SP})$. From now on, a signature name (or specification name, etc.) surrounded by a hat shall mean "constrained."

5 Fundamental results

This section deals with initiality results for positive conditional label specifications. We show that the classical results of [GTW78] can be extended to the framework of label algebras. They are proved for constrained label specifications. Of course, they remain valid for unconstrained label specifications, since a unconstrained specification SP is a constrained specification $\widehat{SP}$ such that $C = \emptyset$. The important results of this section are mainly the theorems 20, 24 and 33. The other results of this section, and all the proofs, can be skipped in a first reading.[4]

We will first prove the following fundamental technical result.

Theorem 20 : Let $\widehat{SP}$ be a positive conditional $\widehat{\Sigma L}$-specification. Let $\mathcal{X}$ be a $\Sigma\widetilde{L}$-algebra. Let R be a binary relation over X compatible with the sorts of the signature (i.e. R is a subset of $\bigcup_{s \in S} X_s \times X_s$). There is a least $\widehat{SP}$-algebra $\mathcal{Y}$ (according to the preorder relation $\leq$ induced by the label morphisms) such that:

1. $\mathcal{X} \leq \mathcal{Y}$, i.e. there exists a label morphism $h_Y : \mathcal{X} \to \mathcal{Y}$
2. $(\mathcal{Y}, h_Y)$ is compatible with R (i.e. $\forall x, y \in X,\ x\ R\ y \implies h_Y(x) = h_Y(y)$).

Proof : Let F be the family of all couples ($\mathcal{Z}$, $h_Z : \mathcal{X} \to \mathcal{Z}$), where $\mathcal{Z}$ is a $\widehat{SP}$-algebra and h_Z is a label morphism such that $(\mathcal{Z}, h_Z)$ satisfies the conditions (1) and (2) of the theorem. F is not empty because $\mathcal{T}riv$ (with the unique trivial morphism from $\mathcal{X}$ to $\mathcal{T}riv$) clearly belongs to F. Thus, we can consider the $\Sigma\widetilde{L}$-algebra $\mathcal{Y} = (Y, \{l_Y\}_{l \in \widetilde{L}})$ as the quotient algebra of $\mathcal{X}$ defined as follows (h_Y being the quotient $\Sigma\widetilde{L}$-morphism):

- $\forall x, y \in X,\ (\ h_Y(x) = h_Y(y)\ \Leftrightarrow\ (\forall(\mathcal{Z}, h_Z) \in F,\ h_Z(x) = h_Z(y))\ )$
- $\forall l \in \widetilde{L},\ \forall x \in \overline{X},\ (\ \overline{h_Y}(x) \in l_Y\ \Leftrightarrow\ (\forall(\mathcal{Z}, h_Z) \in F,\ \overline{h_Z}(x) \in l_Z)\ )$

Let us remark that Y is clearly a Σ-algebra and h_Y is clearly a Σ-morphism (the compatibility with the operations of Σ results from the one of all the elements $\mathcal{Z}$ such that

[4] We apologize to the reader for such tedious formalities !

$(\mathcal{Z}, h_Z)$ belongs to F). Let us also remark that, for the same reason, h_Y is compatible with the labels of $\widetilde{L}$. Thus, h_Y is a $\Sigma\widetilde{L}$-morphism from $\mathcal{X}$ to $\mathcal{Y}$.

Moreover, for every $\mathcal{Z}$ such that $(\mathcal{Z}, h_Z)$ is in F, there exists a $\Sigma\widetilde{L}$-morphism μ_Z from $\mathcal{Y}$ to $\mathcal{Z}$: it is defined by $\forall x \in X,\ \mu_Z(h_Y(x)) = h_Z(x)$ (μ_Z exists by definition of h_Y, and we have $\mu_Z \circ h_Y = h_Z$). Consequently, if $(\mathcal{Y}, h_Y)$ belongs to F then it is its smallest element and the theorem is proved.

It is trivial from the definition of $(\mathcal{Y}, h_Y)$ that it satisfies the conditions (1) and (2) of the theorem. Thus it is sufficient to prove that $\mathcal{Y}$ satisfies $\widehat{SP}$. It remains therefore to prove that $\mathcal{Y}$ satisfies the partial evaluation constraint with respect to $\widehat{L}$ and that it satisfies each axioms of $\widehat{SP}$. It is the purpose of the next two lemmas. □

Lemma 21 : $\mathcal{Y}$ (as defined in the proof of Theorem 20) satisfies the partial evaluation constraint with respect to $\widehat{L}$.

Proof : Let l be a label of C. Let t be a term of $\overline{Y}$. Let u be a subterm of t. By definition of $\mathcal{Y}$, if $t \in l_Y$ then: $\forall \mathcal{Z} \in F,\ \overline{\mu_Z}(t) \in l_Z$. Moreover we have:[5]

$$\overline{\mu_Z}(\ t[u \leftarrow eval_Y(u)]\ ) = \overline{\mu_Z}(t)[\ \overline{\mu_Z}(u) \leftarrow \overline{\mu_Z}(eval_Y(u))\ ] = \overline{\mu_Z}(t)[\ \overline{\mu_Z}(u) \leftarrow eval_Z(u)\ ]$$

Consequently, since all $\mathcal{Z}$ such that $(\mathcal{Z}, h_Z) \in F$ satisfy the partial evaluation constraint, we have:

$$\forall \mathcal{Z} \in F,\ \overline{\mu_Z}(t[u \leftarrow eval_Y(u)]) \in l_Z$$

By definition of $\mathcal{Y}$ this proves that $t[u \leftarrow eval_Y(u)] \in l_Y$. Which proves the lemma. □

Lemma 22 : $\mathcal{Y}$ (as defined in the proof of Theorem 20) satisfies each axiom of $\widehat{SP}$.

Proof : Let $(\ a_1 \wedge \ldots \wedge a_n\ \Rightarrow\ a\ )$ be an axiom of $\widehat{SP}$ (a_i and a being positive atoms). Let $\sigma : V \rightarrow \overline{Y}$ be any assignment covering all the variables of the axiom. By definition of $\mathcal{Y}$, we have:

$$(\forall i = 1..n,\ \mathcal{Y} \models \sigma(a_i)) \iff (\forall (\mathcal{Z}, h_Z) \in F,\ (\forall i = 1..n,\ (\mathcal{Z} \models \overline{\mu_Z}(\sigma(a_i)))))$$

Since all $\mathcal{Z}$ such that $(\mathcal{Z}, h_Z) \in F$ satisfy $\widehat{SP}$, it comes:

$$(\forall i = 1..n,\ \mathcal{Y} \models \sigma(a_i)) \implies (\forall \mathcal{Z} \in F,\ \mathcal{Z} \models \overline{\mu_Z}(\sigma(a)))$$

By definition of $\mathcal{Y}$, we get:

$$(\forall i = 1..n,\ \mathcal{Y} \models \sigma(a_i)) \implies \mathcal{Y} \models \sigma(a)$$

and we obtain that $\mathcal{Y}$ satisfies the axiom under consideration. It proves the lemma, and concludes the proof of the theorem 20. □

The following lemma shows a universal property of $\mathcal{Y}$.

[5] The reader should understand "the occurrence of u" (resp. of $\overline{\mu_Z}(u)$) in t (resp. in $\overline{\mu_Z}(t)$) instead of "u" (resp. $\overline{\mu_Z}(u)$).

Lemma 23 : With the notations of Theorem 20, for every $\widehat{SP}$-algebra $\mathcal{Z}$ satisfying the conditions (1) and (2), there exists a unique morphism $\mu_Z : \mathcal{Y} \to \mathcal{Z}$ such that $\mu_Z \circ h_Y = h_Z$.

Proof : Its existence has already been proved; its unicity results from the surjectivity of h_Y. □

Theorem 24 : Let $\widehat{SP}$ be a positive conditional label specification. $Alg_{Lbl}(\widehat{SP})$ and $Gen_{Lbl}(\widehat{SP})$ have an initial object, denoted $\mathcal{T}_{\widehat{SP}}$.

Moreover, $\mathcal{T}riv$ is final in $Alg_{Lbl}(\widehat{SP})$ (and in $Gen_{Lbl}(\widehat{SP})$ if the signature is sensible).

Proof : The assertion about $\mathcal{T}riv$ is trivial.

The label algebra $\mathcal{T}_{\widehat{SP}}$ is obtained by applying Theorem 20 with $\mathcal{X} = \mathcal{T}_{\Sigma\widetilde{L}}$ and $\mathcal{T}_{\widehat{SP}} = \mathcal{Y}$, R being the empty binary relation. □

The purpose of the remainder of this subsection is to study *structured* positive conditional label specifications. We first define the *forgetful functor* U associated with a structured presentation, then we define the *synthesis functor* F, and we prove that F is left adjoint to U.

Definition 25 : Let $\widehat{\Sigma L_1}$ and $\widehat{\Sigma L_2}$ be two label signatures such that $\widehat{\Sigma L_1} \subset \widehat{\Sigma L_2}$ (i.e. $S_1 \subset S_2$, $\Sigma_1 \subset \Sigma_2$, $L_1 \subset L_2$ and $C_1 \subset C_2$). The forgetful functor U from $Alg_{Lbl}(\widehat{\Sigma L_2})$ to $Alg_{Lbl}(\widehat{\Sigma L_1})$ is defined as follows:

- for each $\widehat{\Sigma L_2}$-algebra $\mathcal{A}$, $U(\mathcal{A})$ is the $\widehat{\Sigma L_1}$-algebra $\mathcal{B}$ defined by:
  1. $\forall s \in S_1,\ B_s = A_s$
  2. $\forall l \in \widetilde{L_1},\ l_B = l_A \cap \overline{B}$
  3. $\forall f \in \Sigma_1,\ f_B = f_A$
- for each $\widehat{\Sigma L_2}$-morphism $\mu : \mathcal{A} \to \mathcal{A}'$, $U(\mu) : U(\mathcal{A}) \to U(\mathcal{A}')$ is the $\widehat{\Sigma L_1}$-morphism μ restricted to $\mathcal{B} = U(\mathcal{A})$ and co-restricted to $\mathcal{B}' = U(\mathcal{A}')$.

Remarks 26 :

1. $\mathcal{B} = U(\mathcal{A})$ satisfies the partial evaluation constraint with respect to $\widehat{L_1}$ because $eval_B$ is equal to $eval_A$ restricted to $\overline{B}$ and corestricted to B. More precisely, let l be a label of C_1 and let t be a term of $\overline{B}$ such that $t \in l_B$. Let u be a subterm of t. By definition of $\mathcal{B}$, we have $t \in l_A$; consequently we have $t[u \leftarrow eval_A(u)] \in l_A$, as $l \in C_2$. Since $eval_A$ and $eval_B$ coincide on $\overline{B}$, we obtain, by definition of l_B, that $t[u \leftarrow eval_B(u)] \in l_B$.
2. $\overline{U(\mu)}$ clearly preserves the labels of $\widetilde{L_1}$; thus $U(\mu)$ is actually a $\widehat{\Sigma L_1}$-morphism.

Theorem 27 : Let $\widehat{SP}_1$ and $\widehat{SP}_2$ be two label specifications such that $\widehat{SP}_1 \subset \widehat{SP}_2$. Let U be the forgetful functor from $Alg_{Lbl}(\widehat{\Sigma L_2})$ to $Alg_{Lbl}(\widehat{\Sigma L_1})$. The restriction of U to $Alg_{Lbl}(\widehat{SP}_2)$ can be co-restricted to $Alg_{Lbl}(\widehat{SP}_1)$.

More generally, given two signatures $\widehat{\Sigma L_1} \subset \widehat{\Sigma L_2}$, for all $\widehat{\Sigma L_2}$-algebras $\mathcal{A}$ and for all $\widehat{\Sigma L_1}$-sentences φ we have:

$$\mathcal{A} \models \varphi \quad \Longrightarrow \quad U(\mathcal{A}) \models \varphi$$

Proof : Let V be the set of variables of φ. We have to prove:

$$(\forall \sigma : V \rightarrow \overline{A},\ \mathcal{A} \models \sigma(\varphi)\ ) \implies (\forall \sigma : V \rightarrow \overline{U(A)},\ U(\mathcal{A}) \models \sigma(\varphi)\ )$$

Since $\overline{U(A)}$ is included in $\overline{A}$ and the labels are preserved, this implication is trivial. □

Remark 28 : Theorem 27 never requires for the sentence φ to be positive conditional. In particular $\widehat{SP_1}$ and $\widehat{SP_2}$ are not necessarily positive conditional specifications.

Let us remark that the reverse implication of Theorem 27 is not valid in general, as shown in the following example. Consequently, the so-called "satisfaction condition" does not hold for label algebras; the framework of label algebras is not an institution (see [GB84]), at least with the natural definitions of signature morphisms and sentence translations.

Example 29 : Let $\widehat{\Sigma L_1}$ be the label signature defined by $S_1 = \{\ \mathit{thesort}\ \}$, $\Sigma_1 = \{\ \mathit{cst}1 : \rightarrow \mathit{thesort}\ \}$ and $L_1 = \{\ \mathit{thelabel}\ \}$ (it does no matter if *thelabel* is constrained or not in this example). Let $\widehat{\Sigma L_2}$ be the label signature defined by $S_2 = S_1$, $\Sigma_2 = \{\ \mathit{cst}1 : \rightarrow \mathit{thesort}\ ,\ \mathit{cst}2 : \rightarrow \mathit{thesort}\ \}$ and $L_2 = L_1$. We clearly have $\widehat{\Sigma L_1} \subset \widehat{\Sigma L_2}$. Let $\mathcal{A}$ be the $\widehat{\Sigma L_2}$-algebra defined by $A = \{a = \mathit{cst}1_A = \mathit{cst}2_A\}$ (A is a singleton) and $\mathit{thelabel}_A = \{\ a\ ,\ \mathit{cst}1\ \}$ (let us remind that $T_{\Sigma_2}(A) = \{\ a\ ,\ \mathit{cst}1\ ,\ \mathit{cst}2\ \}$). The $\widehat{\Sigma L_1}$-algebra $U(\mathcal{A})$ is then characterized by $U(A) = \{a = \mathit{cst}1_{U(A)}\}$ and $\mathit{thelabel}_{U(A)} = \{\ a\ ,\ \mathit{cst}1\ \}$; thus, $\mathit{thelabel}_{U(A)} = T_{\Sigma_1}(U(A))$. Consequently, $U(\mathcal{A})$ satisfies the $\widehat{\Sigma L_1}$-sentence "$x \in \mathit{thelabel}$" while $\mathcal{A}$ does not (as $\mathit{cst}2$ does not belong to $\mathit{thelabel}_A$).

The following technical notation defines a free algebra which will be useful to define the synthesis functor.

Notation 30 : Let $\widehat{\Sigma L_1}$ and $\widehat{\Sigma L_2}$ be two label signatures such that $\widehat{\Sigma L_1} \subset \widehat{\Sigma L_2}$. Let $\mathcal{A}$ be a $\widehat{\Sigma L_1}$-algebra. The $\Sigma \widetilde{L_2}$-algebra $\mathcal{T}_{\Sigma_2}(\mathcal{A}) = (T_{\Sigma_2}(A), \{l_{T_{\Sigma_2}(A)}\}_{l \in \widetilde{L_2}})$ is defined as follows:

1. $T_{\Sigma_2}(A)$ is the usual free Σ_2-term algebra with variables in A.
2. Let $\alpha : A \rightarrow T_{\Sigma_2}(A)$ be the inclusion of A into $T_{\Sigma_2}(A)$.
   Let $\overline{\alpha} : T_{\Sigma_1}(A) \rightarrow T_{\Sigma_1}(T_{\Sigma_2}(A))$ be the canonical Σ_1-morphism which extends α. Let $\iota : T_{\Sigma_1}(T_{\Sigma_2}(A)) \rightarrow T_{\Sigma_2}(T_{\Sigma_2}(A))$ be the canonical inclusion deduced from the inclusion $\Sigma_1 \subset \Sigma_2$. Let (finally) $i : T_{\Sigma_1}(A) \rightarrow T_{\Sigma_2}(T_{\Sigma_2}(A))$ be the composition of $\overline{\alpha}$ and ι.
   For every label $l \in \widetilde{L_1}$, the set $l_{T_{\Sigma_2}(A)}$ is the subset of $T_{\Sigma_2}(T_{\Sigma_2}(A))$ defined by $l_{T_{\Sigma_2}(A)} = i(l_A)$.

3. For every label $l \in (\widetilde{L_2} - \widetilde{L_1})$, $l_{T_{\Sigma_2}(A)}$ is empty.

Definition 31 : Let $\widehat{SP_1}$ and $\widehat{SP_2}$ be two positive conditional label specifications such that $\widehat{SP_1} \subset \widehat{SP_2}$. Let $\mathcal{A}$ be a $\widehat{SP_1}$-algebra and let us consider the $\Sigma\widetilde{L_2}$-algebra $\mathcal{T}_{\Sigma_2}(\mathcal{A})$ defined in Notation 30. Let R be the binary relation defined on $T_{\Sigma_2}(A)$ by:

$$\forall t, t' \in T_{\Sigma_2}(A),\ (t\ R\ t') \Leftrightarrow (t \in T_{\Sigma_1}(A)) \ \wedge \ (t' \in T_{\Sigma_1}(A)) \ \wedge \ (eval_A(t) = eval_A(t'))$$

By definition, $F(\mathcal{A})$ is the least $\widehat{SP_2}$-algebra such that:

1. $\mathcal{T}_{\Sigma_2}(\mathcal{A}) \leq F(\mathcal{A})$ (i.e. there exists a label morphism $h_A : \mathcal{T}_{\Sigma_2}(\mathcal{A}) \rightarrow F(\mathcal{A})$)
2. $(F(\mathcal{A}), h_A)$ is compatible with R.

($F(\mathcal{A})$ exists, from Theorem 20.)

Theorem 32 : With the notations of the previous definition, for each $\widehat{SP_1}$-morphism $\nu : \mathcal{A} \rightarrow \mathcal{A}'$, the $\widehat{SP_2}$-morphism $F(\nu)$ is defined as follows:

- let $\overline{\nu}$ be the canonical $\Sigma\widetilde{L_2}$-morphism from $\mathcal{T}_{\Sigma_2}(\mathcal{A})$ to $\mathcal{T}_{\Sigma_2}(\mathcal{A}')$ deduced from ν. Let $h = h_{A'} \circ \overline{\nu}$ (from $\mathcal{T}_{\Sigma_2}(\mathcal{A})$ to $F(\mathcal{A}')$).
- $(F(\mathcal{A}'), h)$ satisfies the conditions (1) and (2) with respect to $\mathcal{A}$. Consequently there exists a unique morphism $\mu_{F(\mathcal{A}')} : F(\mathcal{A}) \rightarrow F(\mathcal{A}')$ such that $h = \mu_{F(\mathcal{A}')} \circ h_A$ (cf. Lemma 23).
- By definition, $F(\nu) = \mu_{F(\mathcal{A}')}$.

Then, F is a functor from $Alg_{Lbl}(\widehat{SP_1})$ to $Alg_{Lbl}(\widehat{SP_2})$.

Proof : We have to show that $F(\nu' \circ \nu) = F(\nu') \circ F(\nu)$ for all $\widehat{SP_1}$-morphisms $\nu' : \mathcal{A}' \rightarrow \mathcal{A}''$ and $\nu : \mathcal{A} \rightarrow \mathcal{A}'$. This directly results from $\overline{\nu' \circ \nu} = \overline{\nu'} \circ \overline{\nu}$ and from the unicity of the morphism $\mu_F(\mathcal{A}'') : F(\mathcal{A}) \rightarrow F(\mathcal{A}'')$, which is by definition equal to $F(\nu' \circ \nu)$. □

Theorem 33 : Let $\widehat{SP_1}$ and $\widehat{SP_2}$ be two positive conditional label specifications such that $\widehat{SP_1} \subset \widehat{SP_2}$.

The synthesis functor $F : Alg_{Lbl}(\widehat{SP_1}) \rightarrow Alg_{Lbl}(\widehat{SP_2})$ is a left adjoint for the forgetful functor $U : Alg_{Lbl}(\widehat{SP_2}) \rightarrow Alg_{Lbl}(\widehat{SP_1})$.

Proof : Let $\mathcal{A}$ be a $\widehat{SP_1}$-algebra. Let $\alpha : A \rightarrow T_{\Sigma_2}(A)$ be the inclusion of A into $T_{\Sigma_2}(A)$. Let $I_A : A \rightarrow F(\mathcal{A})$ denote the composition of α and h_A. Let us remark that I_A can be corestricted to $U(F(\mathcal{A}))$, as A only contains values of sort belonging to S_1. Since h_A is compatible with the relation R (as defined in Definition 31), I_A is compatible with the operations of Σ_1. Thus, I_A is a Σ_1-morphism from A to $U(F(\mathcal{A}))$. Moreover, from the definition of $\mathcal{T}_{\Sigma_2}(\mathcal{A})$ (Notation 30), I_A is compatible with the labels of $\widetilde{L_1}$. Consequently I_A is in fact a $\Sigma\widetilde{L_1}$-morphism from $\mathcal{A}$ to $U(F(\mathcal{A}))$.

From the Yoneda lemma [McL71], it is sufficient to prove that $(F(\mathcal{A}), I_A)$ is a universal arrow to the forgetful functor U. This means that we have to prove that for all $\widehat{SP_2}$-algebras $\mathcal{B}$ and all $\widehat{SP_1}$-morphism $\eta : \mathcal{A} \rightarrow U(\mathcal{B})$, there exists a unique $\widehat{SP_2}$-morphism $\eta' : F(\mathcal{A}) \rightarrow \mathcal{B}$ such that $\eta = U(\eta') \circ I_A$.

Let us first remark that there exists a unique $\widehat{\Sigma L_2}$-morphism $h_B : \mathcal{T}_{\Sigma_2}(\mathcal{A}) \to \mathcal{B}$ which extends η. Moreover, $(\mathcal{B}, h_B)$ satisfies the conditions (1) and (2) of Definition 31. From Lemma 23, there exists a unique $\widehat{\Sigma L_2}$-morphism η' from $F(\mathcal{A})$ to $\mathcal{B}$ such that $h_B = \eta' \circ h_A$.

It comes $h_B \circ \alpha = \eta' \circ h_A \circ \alpha$. Since h_B is an extension of η, and $h_A \circ \alpha$ an extension of I_A, this equality contains our result: $\eta = U(\eta') \circ I_A$. Moreover, any other morphism ρ satisfying $\eta = U(\rho) \circ I_A$ is then such that $\rho \circ h_A$ is an extension of η. But h_B is the unique extension of η, thus $\rho \circ h_A = h_B$. Finally, the unicity of η' (i.e. $\eta' = \rho$) results from Lemma 23. □

Remark 34 : (For experienced readers. . .) We have shown in this subsection that the framework of label algebras does not form an institution [GB84], even if restricted to positive conditional sentences (cf. Example 29). Indeed, we have proved that the framework of positive conditional label algebras form a specification logic which has free constructions [EBO91][EBCO91]. We imposed an unnecessary restriction: the considered specification morphisms are only inclusions. In particular renaming and non-injective signature morphisms have not been dealt with. We have been motivated by a pedagogical approach. We believe that the reader would had little chance of understanding some of our technical definitions (in particular Notation 30) if the signature morphisms had been explicit.

Let us point out that the specification logic of label algebras has *not* amalgamations (as defined in [EBCO91]). The reason *a priori* is that we have shown in Section 4.2 that observational semantics can be reflected within label algebras, and [EBCO91] has proved that observational semantics have not amalgamations in general. It is the same for extensions (at least if we do not restrict the definition of morphisms).

6 Exception algebras

The framework of exception algebras is a specialization of the one of label algebras, where the labels are used for exception handling purposes.

6.1 Label algebras and exception algebras

As already explained, the normal cases and the exceptional ones must be distinguished without any ambiguity in an exception algebra. A particular label will be distinguished to characterize the normal cases. As in almost all the frameworks about algebraic specifications with exception handling, it will be named Ok. Moreover, exception names and error messages shall be reflected by labels (of course, distinct from Ok). This allows us to take exception names into account in the (label) axioms; thus, an extremely wide spectrum of exception handling and error recovery cases can be specified. Intuitively, when $t \in l_A$ in an exception algebra $\mathcal{A}$ for $l \neq Ok$, it will mean that the calculus defined by t leads to the exception name l; if $l = Ok$, it will mean that the calculus defined by t is a "normal" calculus (i.e. it does not need an exceptional treatment and the calculus is successful).

Most of the time, if $t \in Ok$ then all its subterms are labeled by Ok and lead to Ok-values.[6]

As shown in Section 3, when specifying a data structure with exception handling features, the specifier has first to declare the desired Ok-domain. For instance the interval $[0 \ldots Maxint]$ can be declared as follows:

$$succ^{Maxint}(0) \mathrel{\epsilon} Ok$$

$$succ(n) \mathrel{\epsilon} Ok \implies n \mathrel{\epsilon} Ok$$

(where $succ^{Maxint}(0)$ stands for $succ(succ(\ldots(succ(0))..))$, the operation $succ$ being applied $Maxint$ times.) Let us assume that the specification contains the following "normal axiom:"

$$pred(succ(n)) = n$$

Then, for example, the term $pred(succ(0))$ should also belong to the Ok-domain because its calculus does not require any exceptional treatment and leads to the Ok-term 0 via the previous normal axiom. We have shown in Section 3 that the terseness criteria is not fulfilled when we explicitly describe all the normal terms in an exhaustive manner. Thus, labelling by Ok must be implicitly propagated through the axioms kept for normal cases. These axioms will be called Ok-axioms, and this implicit propagation rule will be an important component of their semantics, as described in Section 7.2. Consequently, the semantics of exception specifications must be more elaborated than the semantics of label specifications; as label algebras have no implicit aspects.

Another important implicit aspect required by exception handling is the so-called "common future" property. Let us assume that $\mathcal{A}$ is an algebra such that a term u has the same value than a term v (i.e. $eval_A(u) = eval_A(v)$; e.g. u can be an exceptional term recovered on the Ok-term v). We clearly need that for every operation f of the signature, $f(\ldots, u, \ldots)$ behaves exactly as $f(\ldots, v, \ldots)$ does. This means that $f(\ldots, u, \ldots)$ and $f(\ldots, v, \ldots)$ must have the same value and raise the same exception names. For example, let $\mathcal{A}$ reflect the natural numbers bounded by $Maxint$; the terms $succ^i(0)$ with $0 \le i \le Maxint$ being labelled by Ok. Let us assume that $succ^{Maxint+1}(0)$ is recovered on $succ^{Maxint}(0)$. Intuitively, once this recovering is done, we want that everything happens as if $succ^{Maxint+1}(0)$ were never raised; this is the very meaning of the word recovering. The recovery should simply work as a systematic replacement of $succ^{Maxint+1}(0)$ by $succ^{Maxint}(0)$. The same succession of operations applied to $succ^{Maxint}(0)$ or to $succ^{Maxint+1}(0)$ should always give the same results; it should return the same value and raise exactly the same exception names. For example if $succ^{Maxint+1}(0)$ is labelled by $TooLarge$ then the term $t = succ^{Maxint+2}(0)$ should also be labelled by $TooLarge$, as $succ^{Maxint+1}(0) = t[succ^{Maxint+1}(0) \leftarrow succ^{Maxint}(0)]$.

Let us note that, in a label algebra $\mathcal{A}$, $eval_A(u) = eval_A(v)$ implies that $eval_A(f(\ldots, u, \ldots))$ is equal to $eval_A(f(\ldots, v, \ldots))$, but it does not imply that the terms $f(\ldots, u, \ldots)$ and $f(\ldots, v, \ldots)$ have the same labels. The common future property means more generally that, for every term t containing u as strict subterm, the term $t[u \leftarrow v]$ has the same exception labels than t. This property is called "common future" and will be an important implicit aspect of the semantics of exception algebras.

[6]except for certain data structures such as intervals which do not contain 0, see Section 9.2.

6.2 Exception signature

Definition 35 : An *exception signature* ΣExc is a label signature $< S, \Sigma, L >$ such that neither Ok nor Exc and Err belong to L. The elements of L are called *exception labels*.

The labels Ok, Exc and Err are not allowed as exception labels because they will be used to characterize the Ok-terms, exceptional terms and erroneous terms respectively.

Example 36 : $NatExc = < \{Nat\}, \{0, succ_, pred_\}, \{Negative, TooLarge\} >$ is a possible exception signature for an exception specification of bounded natural numbers.

As motivated in Section 6.1, exception algebras over the signature ΣExc cannot be directly defined as label algebras over the same signature. We must add the label Ok and we also have to reflect the "common future" property.

Notation 37 : Let $\Sigma Exc = < S, \Sigma, L >$ be an exception signature. In the sequel of this paper, $\widetilde{L}$ will denote $L \cup \{Ok\}$. Moreover $\Sigma\widetilde{L}$ will be the label signature $< S, \Sigma, \widetilde{L} >$. (We will show in Section 7.3 that these notations cope with the notations already introduced in Definition 14.)

Definition 38 : Let $\mathcal{A}$ be a label algebra. Let u be a term of $\overline{A}$. The *future of* u, denoted by $future(u)$, is the set of all the terms $t \in \overline{A}$ that contain u as a *strict* subterm (where "strict" means that u is a subterm of t distinct from t.)

Let us remark that if u and v are terms of the same sort then the following property holds:

$$future(v) = \{ \ t[u \leftarrow v] \mid t \in future(u) \ \}$$

Moreover, if $eval_A(u) = eval_A(v)$ then we have:

$$\forall t \in future(u) \ , \ eval_A(t[u \leftarrow v]) = eval_A(t)$$

Definition 39 : Let $\mathcal{A}$ be a label algebra and let l be a label of the underlying label signature. By definition $\mathcal{A}$ satisfies the *common future property* for l if and only if for all terms u and v of $\overline{A}$ such that $eval_A(u) = eval_A(v)$ we have:

$$\forall t \in future(u) \ , \ t \in l_A \ \Leftrightarrow \ t[u \leftarrow v] \in l_A$$

Theorem 40 : $\mathcal{A}$ satisfies the common future property for l if and only if for all operations $f : s_1 \ldots s_n \to s$ of the signature with $n > 0$, and all terms $u_1 \ldots u_n$ and $v_1 \ldots v_n$ of $\overline{A}$ (of sort $s_1 \ldots s_n$ respectively), we have:

$$\left(\bigwedge_{i=1}^{n} eval_A(u_i) = eval_A(v_i) \ \right) \ \wedge \ f(u_1, \ldots, u_n) \in l_A \ \Longrightarrow \ f(v_1, \ldots, v_n) \in l_A$$

Proof : Immediate, by recurrence on the size of the terms. □

Definition 41 : An *exception algebra* over the exception signature ΣExc is a label algebra $\mathcal{A}$ over the signature $\Sigma\widetilde{L}$ that satisfies the common future property for every l in L.

This definition call for some comments:

- We have carefully excluded the term u from $future(u)$ by considering only strict subterms. If we accept $t = u$ as a future of u then it amounts to directly labelling values, because the common future property becomes equivalent to:

$$\forall t, t' \in \overline{A}\ ,\ \ eval_A(t) = eval_A(t') \implies (t \in l_A \Leftrightarrow t' \in l_A)$$

  and everything happens exactly as if labelling were attached to values. Consequently our semantics would be equivalent to the ones of [BBC86] and we have shown in Example 2 that this is not suitable. Precisely, the common future property is a weaker constraint than the labelling of values, and it ensures a significant difference; for instance Example 2 get the suitable initial model (see also Section 9).

- The label Ok must not be concerned with the common future property. Otherwise, if $succ(Maxint)$ is recovered on $Maxint$, we would have that $pred(succ(Maxint))$ is labelled by Ok ($pred(Maxint)$ being labelled by Ok). Clearly, even if the term $pred(succ(Maxint))$ is recovered, it must remain exceptional because an exceptional treatment has been required in its history. The axiom $pred(succ(x)) = x$ being a normal axiom; if $pred(succ(Maxint))$ is not considered as exceptional then the assignment $x = Maxint$ would be an acceptable assignment and we would have the inconsistency $pred(Maxint) = Maxint$ (see also Section 7.2).

- Notice that the common future property implies that the labelling of a term t by an exception label mainly relies on the heading symbol of t. More precisely, for every operation f of the signature, if we have $eval_A(u_i) = eval_A(v_i)$ for all i in $\{1, \ldots, n\}$, then $f(u_1, \ldots, u_n)$ and $f(v_1, \ldots, v_n)$ carry the same exception labels. Consequently, for every operation f, we can inventory the set of labels that can be raised by f. It reflects the classical exception declarations of programming languages with exception handling, such as CLU or ADA (see for instance [LG86]), where a given function can raise only a subset of the exception names, which are declared in the heading of the function.

Example 42 : According to the exception signature $NatExc$ define above, we can consider for example the exception algebra $\mathcal{A} = (A, \{l_A\}_{l\in\widetilde{L}})$ defined by:

$$A = \{\ldots, -2, -1, 0, 1, 2, \ldots, Maxint\}$$

the operations $succ_A$ and $pred_A$ are defined as usual on integers with the restriction $succ_A(Maxint) = Maxint$

$Negative_A =$

$$\left\{\begin{array}{lllllll} \ldots, & \ldots, & pred(pred(0)), & pred(0), & succ(pred(0)), & & \ldots, \\ \ldots, & -3, & -2, & -1, & succ(-1), & succ(succ(-1)), & \\ \ldots, & \ldots, & succ(-3), & succ(-2), & succ(succ(-2)), & & \ldots, \end{array}\right\}$$

$Negative_A$ contains here at the same time negative values and terms. All these terms have a negative value by classical evaluation in the set of integer or else have at least a subterm which would have a negative value by evaluation.

$$TooLarge_A = \{succ^{Maxint+1}(0), succ(Maxint), succ(succ(Maxint)), \ldots\}$$

$$Ok_A = \left\{ \begin{array}{rrrr} \ldots, & succ(0), & succ(1), & \ldots, \\ 0, & 1, & 2, & 3, \\ pred(1), & pred(2), & pred(3), & \ldots, \end{array} \right\}$$

Definition 43 : Let $\mathcal{A}$ and $\mathcal{B}$ be two exception algebras with respect to the exception signature ΣExc. An exception morphism $\mu : \mathcal{A} \to \mathcal{B}$ is a $\Sigma\widetilde{L}$-morphism from $\mathcal{A}$ to $\mathcal{B}$.

Remark 44 : We accept ΣExc-signature, ΣExc-algebra, ΣExc-morphism as additional notations respectively for exception signature, exception algebra and exception morphism with respect to the exception signature ΣExc.

Definition 45 : Given an exception signature ΣExc, the category of all ΣExc-algebras, and ΣExc-morphisms, is denoted by $Alg_{Exc}(\Sigma Exc)$.

Theorem 46 : $Alg_{Exc}(\Sigma Exc)$ has an initial object denoted $\mathcal{T}_{\Sigma Exc}$.

Proof : $Alg_{Exc}(\Sigma Exc)$ is included in $Alg_{Lbl}(\Sigma\widetilde{L})$ and the initial object of $Alg_{Lbl}(\Sigma\widetilde{L})$, $\mathcal{T}_{\Sigma\widetilde{L}}$, satisfies the common future property. Thus $\mathcal{T}_{\Sigma\widetilde{L}}$, also denoted $\mathcal{T}_{\Sigma Exc}$, is initial in $Alg_{Exc}(\Sigma Exc)$. □

Definition 47 : $Gen_{Exc}(\Sigma Exc)$ is the full subcategory of $Alg_{Exc}(\Sigma Exc)$ containing all the finitely generated algebras.

Theorem 48 : Let ΣExc be an exception signature. Let $Fut_{\Sigma,L}$ be the positive conditional label specification which contains all the $\Sigma\widetilde{L}$-axioms of the form:

$$x_1 = y_1 \ \wedge \ldots \wedge \ x_n = y_n \ \wedge \ f(x_1, \ldots, x_n) \,\epsilon\, l \implies f(y_1, \ldots, y_n) \,\epsilon\, l$$

where f is any non-constant operation of Σ (i.e. $n > 0$), x_i and y_i are variables of sorts given by the arity of f, and l is any exception label of L. (If there are p non-constant operations in Σ and q exception labels in L, then $Fut_{\Sigma,L}$ contains $p \times q$ axioms.)

The label specification $Fut_{\Sigma,L}$ specifies the ΣExc-algebras, i.e. $Alg_{Exc}(\Sigma Exc)$ is equal to $Alg_{Lbl}(Fut_{\Sigma,L})$.

Proof : From Definition 12 and Theorem 40, a $\Sigma\widetilde{L}$-algebra $\mathcal{A}$ satisfies $Fut_{\Sigma,L}$ if and only if it satisfies the common future property. Thus, $Alg_{Exc}(\Sigma Exc) = Alg_{Lbl}(Fut_{\Sigma,L})$. □

6.3 Exceptions and errors

By analogy with programming languages, a term is exceptional if and only if it raises and exception name in its history.

Definition 49 : Let $\mathcal{A}$ be a ΣExc-algebra. The set of *exceptional terms* according to $\mathcal{A}$ is the least subset of $\overline{A}$, denoted by Exc_A, such that:

1. for all labels $l \in L$, $l_A \subset Exc_A$;
2. for all t in Exc_A, $future(t) \subset Exc_A$.

In other words, $Exc_A = \bigcup_{l \in L} future(l_A)$. Notice that, of course, $l = Ok$ is not taken into account in this definition.

Let us remark that an exceptional term is not necessarily erroneous because it can be recovered on an Ok-value. Nevertheless it remains exceptional because its recovery has required an exceptional treatment.

Definition 50 : Let $\mathcal{A}$ be a ΣExc-algebra. The set of *Ok-values* of A is defined by $A_{Ok} = eval_A(Ok_A)$.

Notice that Ok_A is a set of terms (subset of $\overline{A}$) while A_{Ok} is a set of values (subset of A). Then, erroneous terms can easily be defined:

Definition 51 : Let $\mathcal{A}$ be a ΣExc-algebra. The set of *erroneous terms* according to $\mathcal{A}$ is the least subset of $\overline{A}$, denoted by Err_A, such that:

1. for all labels $l \in L$, for all terms $t \in l_A$, if $eval_A(t) \notin A_{Ok}$ then $t \in Err_A$; i.e.:
$$[(\bigcup_{l \in L} l_A) - eval_A^{-1}(A_{Ok})] \subset Err_A$$
2. for all non-constant operations $(f : s_1 \ldots s_n \rightarrow s)$ of the signature ($n > 0$), for all terms $t_1, \ldots, tn$ (according to the arity of f), if at least one of the t_i belongs to Err_A and if $eval_A(f(t_1, \ldots, t_n)) \notin A_{Ok}$ then $f(t_1, \ldots, t_n)$ belongs to Err_A

Moreover, the set of erroneous values of A is by definition $A_{Err} = eval_A(Err_A)$.

These definitions call for some comments.

Remarks 52 :

1. By construction of A_{Err}, we have $A_{Ok} \cap A_{Err} = \emptyset$. However, $A_{Ok} \cup A_{Err}$ does not necessarily cover A. A value of A that does neither belong to A_{Ok} nor to A_{Err} is the evaluation of terms that do not raise exceptional name, but have no Ok-value. Intuitively, in this case, it reflects *partial functions*. For example, let us assume that $\mathcal{A}$ is an algebra reflecting natural numbers such that the terms $n\ div\ 0$ are not labelled, but do not get an Ok-value. Then it means that the operation div does not raise an explicit exception, and the division by 0 is undefined.

2. Every erroneous term is exceptional ($Err_A \subset Exc_A$), but the converse is false because an exception can be recovered. However, let us remark that Err_A is not equal to $(Exc_A - Ok_A)$, and also that it is not equal to $(Exc_A - eval_A^{-1}(A_{Ok}))$. More precisely we have:

$$Err_A \subset (Exc_A - eval_A^{-1}(A_{Ok})) \subset (Exc_A - Ok_A)$$

but none of the reverse inclusions is ensured. For example, let us consider an algebra $\mathcal{A}$ reflecting bounded natural numbers where $succ(Maxint) = Maxint$, the terms $n\ div\ 0$ being not labelled (as in the previous example). Then the term $(succ(Maxint)\ div\ 0)$ is exceptional (because $succ(Maxint)$ is exceptional), it is not recovered, but it is not erroneous ($succ(Maxint)$ is not erroneous, since recovered). It is simply equal to $(Maxint\ div\ 0)$, and undefined.

3. Notice that the definitions of Err_A, Exc_A, A_{Ok} and A_{Err} are independent of any specification; they are intrinsically defined from the considered exception algebra $\mathcal{A}$.

4. Of course, we can consider that Exc and Err are new labels, and automatically build a label algebra over the label signature $< S, \Sigma, L \cup \{Ok, Exc, Err\} >$ from any exception algebra. However, we should be aware that *exception morphisms do not preserve the label Err*, because exception morphisms can reflect additional recoveries (see Example 54 below). It is not difficult to show that they preserve the label Exc, i.e. $\overline{\mu}(Exc_A) \subset Exc_B$.

Definition 53 : An exception algebra $\mathcal{A}$ is called *self complete* (or *total*) if and only if $A = A_{Ok} \cup A_{Err}$.

Example 54 : In the exception algebra $\mathcal{A}$ described in the previous section, Example 42, we have for instance:

- $pred(0)$ and all the terms that contain $pred(0)$ as subterm are exceptional because $pred(0)$ belongs to $Negative_A$;
- $succ^{Maxint+1}(0)$ is recovered since it is exceptional and its value is equal to the value of the Ok-term $succ^{Maxint}(0)$;
- $pred(0)$ is an erroneous term since it belongs to $Negative_A$ without belonging to $eval_A^{-1}(A_{Ok})$;
- -1 is an erroneous value since this is the result of the evaluation of the erroneous term $pred(0)$;
- If we consider an algebra $\mathcal{B}$ that additionally recovers $pred(0)$ on 0, there exists an exception morphism from $\mathcal{A}$ to $\mathcal{B}$, which is the quotient morphism, but it does not preserve the label Err, as $pred(0)$ does not belong to Err_B.

7 Exception specifications

As shown in Section 2, it is necessary to separate the axioms concerning exceptional cases from the ones concerning normal cases in order to preserve clarity and terseness of specifications. The axioms of an exception specification will be separated in two parts.

- The first part, called *GenAx*, is mainly devoted to exception handling. It has three main purposes:

  1. We have shown in the sections 3.4 and 3.5 that it is first necessary to characterize the *Ok*-domains of the underlying data structures. They will be specified in *GenAx* by positive conditional axioms with a conclusion of the form $t \epsilon Ok$, whose meaning is that t must be a normal term. Thus, these axioms will be used as starting point to generate the set of *Ok*-terms.
  2. It is also necessary to attach exception names to the exceptional cases, in order to facilitate the specification of specialized exception handlings. They will be specified in *GenAx* by positive conditional axioms with a conclusion of the form $t \epsilon l$ where l belongs to L, whose meaning is that the heading function of the term t raises the exception name l.
  3. The third purpose of *GenAx* is to handle the exceptional cases, in particular to specify recoveries, according to the previous labelling of terms. They will have a conclusion of the form $u = v$.

  As the axioms of *GenAx* concern all the terms, exceptional or not, the satisfaction of such axioms does not require some particular mechanism; it will simply be the same as for label axioms. It is the reason why the three purposes mentioned above are grouped under the name "generalized axioms" (they have common semantics); however, for a concrete syntax devoted to exception specifications, it could be preferable to distinguish these three purposes.

- The second part, called *OkAx*, is entirely devoted to the normals cases, and will only concern terms labelled by *Ok*. As extensively shown in Section 3, the semantics of *OkAx* must be carefully restricted to *Ok*-assignments only, in order to avoid inconsistencies.

  We will define a special semantics for *Ok*-axioms that will both specify equalities between *Ok*-terms and carefully propagate labelling by *Ok* through these equalities (following the motivation given in Section 6.1).

An exception specification *SPEC* is defined as a triple $< \Sigma Exc, GenAx, OkAx >$ where ΣExc is an exception signature, *GenAx* a set of generalized axioms (defined in Section 7.1 below) and *OkAx* a set of *Ok*-axioms (defined in Section 7.2 below).

7.1 Generalized axioms

Definition 55 : Let ΣExc be an exception signature. A set of *generalized axioms* with respect to the exception signature ΣExc is a set *GenAx* of positive conditional label axioms with respect to the label signature $\Sigma\widetilde{L}$.

Definition 56 : Given an exception signature ΣExc, an exception algebra $\mathcal{A}$ satisfies a generalized axiom "α" ($\mathcal{A} \models \alpha$) if and only if the underlying label algebra of $\mathcal{A}$ satisfies it, regarded as a label axiom.

Given a set $GenAx$ of generalized axioms, $\mathcal{A}$ satisfies $GenAx$ if and only if $\mathcal{A}$ satisfies all the axioms of $GenAx$.

Example 57 : Let $NatExc = < \{Nat\}, \{0, succ_, pred_\}, \{TooLarge, Negative\} >$ be the exception signature given in Example 36. An example of $GenAx$ for a specification of natural numbers bounded by $Maxint$ is given by Figure 57.

2— An example of $GenAx$

$succ^{Maxint}(0) \,\epsilon\, Ok$
$succ(n) \,\epsilon\, Ok \Rightarrow n \,\epsilon\, Ok$
$succ^{Maxint+1}(0) \,\epsilon\, TooLarge$
$pred(0) \,\epsilon\, Negative$
$succ^{Maxint+1}(0) = succ^{Maxint}(0)$

The two first axioms specify the Ok domain of Nat. In most examples, they define recursively the set of "normal forms" which belong to the intended Ok domain. It is not necessary to declare *all* the Ok-terms (the label Ok will be automatically be propagated to terms such as $pred(succ(0))$ via the Ok-axioms, as described in Section 7.2). Even if it is generally easier to specify the Ok domain this way, it is not mandatory. We never require for the axioms of an exception specification to be canonical term rewriting systems, and a fortiori, we never require to actually specify normal forms. It is only desirable to declare at least one term for each intended Ok-value.

The third and fourth axioms declare exception names. Their meaning is that the operation *succ* (resp. *pred*) raises the exception $TooLarge$ (resp. $Negative$) when applied to $Maxint$ (resp. 0).

The last axiom recovers $succ^{Maxint+1}(0)$ on $succ^{Maxint}(0)$. Let us note that the generalized axiom $succ^{Maxint+1}(0) \,\epsilon\, TooLarge$ is then not directly necessary, but it could have been useful if we replaced the last axiom by:

$$n \,\epsilon\, TooLarge \implies n = succ^{Maxint}(0)$$

which can make the specification more easily understandable, or by

$$succ(n) \,\epsilon\, TooLarge \implies succ(n) = n$$

which is consistent now, because the label $TooLarge$ is not propagated to $succ^{Maxint}(0)$ (see Example 2). On the other hand, since $pred(0)$ is not recovered, it is necessary to label it in order to write a "self complete" specification (defined in Section 8.1). More precisely, as soon as $pred(0)$ is labelled by $Negative$ and not recovered, it becomes erroneous.

Theorem 58 : Let ΣExc be an exception signature and let $GenAx$ be a set of generalized axioms. The class of exception algebras which satisfies $GenAx$ is equal to $Alg_{Lbl}(Fut_{\Sigma,L} \cup GenAx)$. ($Fut_{\Sigma,L}$ is defined in Theorem 48.)

Proof : The label specification $Fut_{\Sigma,L}$ defined in Theorem 48 exactly specifies the property of common future for L; i.e. a label algebra $\mathcal{A}$ has the common future property w.r.t. L if and only if it satisfies $Fut_{\Sigma,L}$. Moreover, by definition, the satisfaction of generalized axioms is exactly the one of label axioms. Thus, the theorem is immediate. □

7.2 Ok-axioms

Definition 59 : Let ΣExc be an exception signature. A set of *Ok-axioms* with respect to the exception signature ΣExc is a set $OkAx$ of positive conditional $\Sigma\widetilde{L}$-axioms with a conclusion of the form: $v = w$. Thus, an Ok-axiom is of the form

$$u_1 \,\epsilon\, l_1 \;\wedge \ldots \wedge\; u_m \,\epsilon\, l_m \;\wedge\; v_1 = w_1 \;\wedge \ldots \wedge\; v_n = w_n \;\Longrightarrow\; v = w$$

where the u_i, v_j, w_j, v and w are Σ-terms with variables and the l_i are labels of $\widetilde{L}$. (m or n may be equal to 0.)

Definition 60 : Let ΣExc be an exception signature. An exception algebra $\mathcal{A}$ satisfies an Ok-axiom of the form:

$$u_1 \,\epsilon\, l_1 \;\wedge \ldots \wedge\; u_m \,\epsilon\, l_m \;\wedge\; v_1 = w_1 \;\wedge \ldots \wedge\; v_n = w_n \;\Longrightarrow\; v = w$$

if and only if for all assignments σ with range in $\overline{A}$ (covering all the variables of the axiom) which satisfy the precondition, i.e.

$$(\bigwedge_{i=1}^{m} \sigma(u_i) \in l_{iA}) \;\wedge\; (\bigwedge_{j=1}^{n} eval_A(\sigma(v_j)) = eval_A(\sigma(w_j)))$$

the two following properties hold:

1. *Ok propagation*: if at least one of the terms $\sigma(v)$ or $\sigma(w)$ belongs to Ok_A and the other one is of the form $f(t_1, \ldots, t_p)$ with all the t_i belonging to Ok_A (p may be equal to 0), then both $\sigma(v)$ and $\sigma(w)$ belong to Ok_A.
2. *Ok equality*: if $\sigma(v)$ and $\sigma(w)$ belong to Ok_A then $eval_A(\sigma(v)) = eval_A(\sigma(w))$.

$\mathcal{A}$ satisfies $OkAx$ if and only if $\mathcal{A}$ satisfies all the Ok-axioms of $OkAx$.

The semantics of $OkAx$ call for some comments.

1. The first property of the definition reflects a propagation of the Ok label: a term can be labelled by Ok through an Ok-axiom only if all its *direct strict subterms* (i.e. all the arguments of the heading function) are already Ok. This rule allows us to carefully propagate the label Ok. It corresponds to an *innermost evaluation* which avoids inconsistencies: a recovered exceptional term cannot be treated by the

Ok-axioms. Intuitively, an innermost evaluation reflects an implicit propagation of exceptions: if t is not an Ok-term then $f(\ldots,t,\ldots)$ cannot be turned into an Ok-term via the Ok-axioms (recoveries must be handled by generalized axioms because they are exceptional treatments). Thus, $t \in Ok_A$ does not mean "t has an Ok value;" it means "t does not require an exceptional treatment in its history."

Let us note that this propagation starts from the Ok-terms declared in $GenAx$.

2. The second property specifies the equalities that must hold for the normal cases. Two terms have the same evaluation according to an Ok-axiom only if they are both labelled by Ok.

Example 61 : Let $NatExc = < \{Nat\}, \{0, succ_, pred_\}, \{TooLarge, Negative\} >$ and $GenAx$ be given as in Example 57. The set $OkAx$ of Ok-axioms has only to specify the operation *pred* in all normal cases. It can be given by the single following axiom:

$$pred(succ(n)) = n$$

Notice that $OkAx$ is actually terse and clear, compared to the approaches described in Section 3. Notice moreover that the inconsistency raised by the recovery $succ^{Maxint+1}(0) = succ^{Maxint}(0)$ does not occur any more; the instance

$$pred(succ^{Maxint}(0)) = pred(succ^{Maxint+1}(0)) = succ^{Maxint}(0)$$

is no longer an instance of the Ok-axiom because the term $succ^{Maxint+1}(0)$, therefore the term $pred(succ^{Maxint+1}(0))$, is not required to be an Ok-term in our framework (even though $eval_A(succ^{Maxint+1}(0))$ is equal to $eval_A(succ^{Maxint}(0))$). Thus, $pred(succ^{Maxint+1}(0)) = succ^{Maxint}(0)$ is *not* a consequence of $OkAx$. This is a good example of our restricted propagation of the label Ok through the Ok-axioms; it shows how the semantics of Ok-axioms reflects an implicit propagation of exceptions.

Definition 62 : An *exception specification* is a triple $< \Sigma Exc, GenAx, OkAx >$ where ΣExc is an exception signature, $GenAx$ is a set of generalized axioms and $OkAx$ is a set of Ok-axioms.

Let $SPEC =< \Sigma Exc, GenAx, OkAx >$. A ΣExc-algebra $\mathcal{A}$ satisfies $SPEC$ if and only if it satisfies $GenAx$ and $OkAx$, as sets of generalized axioms and Ok-axioms respectively.

We denote by $Alg_{Exc}(SPEC)$ the full subcategory of $Alg_{Exc}(\Sigma Exc)$ containing all the algebras satisfying $SPEC$ ($SPEC$-algebras for short). $Gen_{Exc}(SPEC)$ is the full subcategory of $Alg_{Exc}(SPEC)$ containing the finitely generated $SPEC$-algebras.

Note that the semantics of an axiom with a conclusion of the form $v = w$ vary whether it is considered as a generalized axiom or as an Ok-axiom, according to the principle that the specification of normal cases and exceptional cases must be distinguished, with different implicit semantics.

Remark 63 : Let $BoundedNat$ be the exception specification given in the example above. The exception algebra $\mathcal{A}$ described in Example 42 satisfies $BoundedNat$.

Remark 64 : As a consequence of the semantics of $OkAx$, non-strict operations or lazy-evaluation cannot be specified by means of Ok-axioms. Precisely, the specification of non-strict operations or lazy-evaluation rely on exception handling because it concerns all the terms, even if they contain exceptional subterms, provided that they satisfy a given condition. Consequently, such operations must be specified via generalized axioms. For example, to specify a non-strict $if_then_else_$ operation, the two following axioms have to be put into $GenAx$:

$$if\ true\ then\ u\ else\ v = u$$

$$if\ false\ then\ u\ else\ v = v$$

Being considered as generalized axioms, they concern all the terms $\sigma(u)$ and $\sigma(v)$, including the non-Ok ones. For instance, even if $\sigma(v)$ is erroneous, the term $\sigma(if\ true\ then\ u\ else\ v)$ can result in an Ok-value, as soon as $\sigma(u)$ has an Ok-value.

Let us note that the first "naive" (but terse) algorithm given in Section 2.1 works correctly provided that the operation $_and_$ is a lazy operator which evaluates first the left hand side argument. This lazyness can be specified in $GenAx$ as follows:

$$false\ and\ v = false$$

the other usual axiom ($true\ and\ v = v$) can be left in $OkAx$ for instance.

Lemma 65 : Let ΣExc be an exception signature. Let α be an Ok-axiom. There is a set of $\Sigma\widetilde{L}$-axioms, denoted $Tr(\alpha)$, such that for every ΣExc-algebra $\mathcal{A}$, $\mathcal{A}$ satisfies the Ok-axiom α if and only if the underlying $\Sigma\widetilde{L}$-algebra of $\mathcal{A}$ satisfies $Tr(\alpha)$, regarded as a set of label axioms.

Proof : By definition, the Ok-axiom α is of the form

$$P \implies v = w$$

where P is the precondition of α (P may be empty). Three cases can occur, depending whether the terms v and w are reduced to a variable or not.

1. If v and w are not reduced to variables, then $v = f(v_1, \ldots, v_p)$ and $w = g(w_1, \ldots, w_q)$ where f and g belong to the signature and v_i and w_j are terms with variables (p or q may be equal to 0). Let $Tr(\alpha)$ be the following set of $\Sigma\widetilde{L}$-axioms:

$$P \wedge (\bigwedge_{i=1}^{p} v_i \in Ok) \wedge w \in Ok \implies v \in Ok$$

$$P \wedge v \in Ok \wedge (\bigwedge_{j=1}^{q} w_j \in Ok) \implies w \in Ok$$

$$P \wedge v \in Ok \wedge w \in Ok \implies v = w$$

From Definition 12, the underlying label algebra of an exception algebra $\mathcal{A}$ satisfies the two first label axioms if and only if it satisfies the Ok propagation of Definition 60; moreover $\mathcal{A}$ satisfies the last label axiom if and only if it satisfies the Ok equality of Definition 60. Consequently, $\mathcal{A}$ satisfies α as an Ok-axiom if and only if it satisfies these three axioms as label axioms.

2. If exactly one of the terms v or w is reduced to a variable, say v; then w is is of the form $g(w_1, \ldots, w_q)$ and v is a variable. Let n be the cardinal of the signature (the number of operations belonging to ΣExc). For each operation f of ΣExc, let σ_f be the assignment defined by $\sigma_f(v) = f(z_1, \ldots, z_p)$ where z_i are fresh variables[7] and $\sigma_f(x) = x$ for all the other variables x appearing in α. Let, finally, $Tr(\alpha)$ be the set containing the following $n + 2$ label axioms:

$$\sigma_f(P) \ \wedge \ (\bigwedge_{i=1}^{p} z_i \,\epsilon\, Ok) \ \wedge \ \sigma_f(w) \,\epsilon\, Ok \ \Longrightarrow \ \sigma_f(v) \,\epsilon\, Ok$$

(for all f in the signature) and

$$P \ \wedge \ v \,\epsilon\, Ok \ \wedge \ (\bigwedge_{j=1}^{q} w_j \,\epsilon\, Ok) \ \Longrightarrow \ w \,\epsilon\, Ok$$

$$P \ \wedge \ v \,\epsilon\, Ok \ \wedge \ w \,\epsilon\, Ok \ \Longrightarrow \ v = w$$

Let us note that for each assignment σ with range in $\overline{A}$, either $\sigma(v)$ is a constant element of A, or there exists an assignment γ and a σ_f such that $\sigma = \gamma \circ \sigma_f$. Consequently, for the same reason as before, the underlying label algebra of an exception algebra $\mathcal{A}$ satisfies the $(n + 1)$ first axioms if and only if it satisfies the Ok propagation; moreover it satisfies the last axiom if and only if it satisfies the Ok equality of Definition 60. Consequently, $\mathcal{A}$ satisfies α as an Ok-axiom if and only if it satisfies these $(n + 2)$ axioms as label axioms.

3. If v and w are two distincts variables (if they are equal $Tr(\alpha) = \emptyset$ is sufficient), let $Tr(\alpha)$ be the set containing the following $(2n + 1)$ label axioms:

$$\sigma_f(P) \ \wedge \ (\bigwedge_{i=1}^{p} z_i \,\epsilon\, Ok) \ \wedge \ w \,\epsilon\, Ok \ \Longrightarrow \ \sigma_f(v) \,\epsilon\, Ok$$

(for all f in the signature) and

$$\tau_g(P) \ \wedge \ v \,\epsilon\, Ok \ \wedge \ (\bigwedge_{j=1}^{q} z_j \,\epsilon\, Ok) \ \Longrightarrow \ \tau_g(w) \,\epsilon\, Ok$$

(for all g in the signature, the τ_g being defined with respect to w in a similar manner as the σ_g have been defined with respect to v) and

$$P \ \wedge \ v \,\epsilon\, Ok \ \wedge \ w \,\epsilon\, Ok \ \Longrightarrow \ v = w$$

For the same reasons as before, an exception algebra $\mathcal{A}$ satisfies the Ok-axiom α if and only if it satisfies these $(2n + 1)$ label axioms.

This proves the lemma. □

[7] according to the arity of f; p may be equal to 0.

Theorem 66 : Let $SPEC = < \Sigma Exc, GenAx, OkAx >$ be an exception specification. Let $Tr(SPEC)$ be the label specification defined by the label signature $\Sigma\widetilde{L}$ and the set of label axioms containing: all the axioms of $Fut_{\Sigma,L}$ (defined in Theorem 48), $GenAx$ and all the $Tr(\alpha)$ for $\alpha \in OkAx$ (defined in Lemma 65 above). We have $Alg_{Exc}(SPEC) = Alg_{Lbl}(Tr(SPEC))$. $Tr(SPEC)$ is called the *translation* of the exception specification $SPEC$ into a label specification.

Proof : Directly results from Theorem 58 and Lemma 65. □

Let us note that $Tr(SPEC)$ contains only positive conditional axioms. Section 8.1 uses $Tr(SPEC)$ to obtain initiality results for exception algebras.

7.3 Constrained exception algebras

As already mentioned in Section 4.3, if a term t belongs to Ok_A for an exception algebra $\mathcal{A}$, then any partial evaluation of t should intuitively belong to Ok_A as well. This means that it is not restrictive to turn the label Ok into a *constrained* label (as defined in Section 4.3). We have also shown that, on the contrary, the labels reflecting exception names (those of L) must not be constrained. Thus, the definition of exception algebras can be refined into constrained exception algebras as follows.

Definition 67 : Let $SPEC$ be an exception specification, whose exception signature is $\Sigma Exc = < S, \Sigma, L >$. A *constrained SPEC-algebra* is a $SPEC$-algebra $\mathcal{A}$ such that the underlying label algebra is a constrained label algebra with respect to the constrained label signature $\widehat{\Sigma L} = < S, \Sigma, (L, \{Ok\}) >$ (i.e. $C = \{Ok\}$; constrained label algebras are defined in Definition 17).

Conventionally, we denote $Alg_{Exc}(\widehat{SPEC})$ the full subcategory of $Alg_{Exc}(SPEC)$ containing all the constrained $SPEC$-algebras. $\widehat{SPEC}$ will be called "constrained exception specification," which means that we consider its constrained semantics only. Similar notations hold for $\widehat{\Sigma Exc}$, $Alg_{Exc}(\widehat{\Sigma Exc})$, $Gen_{Exc}(\widehat{\Sigma Exc})$ and $Gen_{Exc}(\widehat{SPEC})$. (These notations are only conventional; from the syntactical point of view, $\widehat{SPEC}$ is indeed equal to $SPEC$.)

Notation 68 : Let $\widehat{SPEC} = < \Sigma Exc, GenAx, OkAx >$ be a constrained exception specification, whose exception signature is $\Sigma Exc = < S, \Sigma, L >$. We denote by $\widehat{Tr}(\widehat{SPEC})$ the constrained label specification defined by the constrained label signature $\widehat{\Sigma L} = < S, \Sigma, (L, \{Ok\}) >$ and the same label axioms as $Tr(SPEC)$ (defined in Theorem 66).

Theorem 66 is easily refined to constrained exception specifications.

Theorem 69 : For all constrained exception specifications $\widehat{SPEC}$, $Alg_{Exc}(\widehat{SPEC})$ is equal to $Alg_{Lbl}(\widehat{Tr}(\widehat{SPEC}))$.

Proof : From Theorem 66, we have $Alg_{Exc}(SPEC) = Alg_{Lbl}(Tr(SPEC))$. Moreover, by definitions, $Alg_{Exc}(\widehat{SPEC})$ is the full subcategory of $Alg_{Exc}(SPEC)$ containing the Ok-constrained algebras, and $Alg_{Lbl}(\widehat{Tr}(\widehat{SPEC}))$ is the full subcategory of $Alg_{Lbl}(Tr(SPEC))$

containing the Ok-constrained algebras. Consequently, $Alg_{Exc}(\widehat{SPEC}) = Alg_{Lbl}(\widehat{Tr}(\widehat{SPEC}))$. □

8 Main results and structured exception specifications

8.1 Fundamental results

The translations proved in Section 7 above ensure that all the initiality results obtained for label algebras hold for exception algebras. In order to simplify the notations, they are stated in this section for unconstrained exception specifications, but they clearly apply to constrained exception specifications, as $\widehat{Tr}(\widehat{SPEC})$ is similar to $Tr(SPEC)$ and all the results of Section 5 are valid for constrained label algebras.

Theorem 70 : Let $SPEC$ be an exception specification. Both $Alg_{Exc}(SPEC)$ and $Gen_{Exc}(SPEC)$ have an initial object, denoted $\mathcal{T}_{SPEC}$.

Moreover, $\mathcal{T}riv$ is final in $Alg_{Exc}(SPEC)$ (and in $Gen_{Exc}(SPEC)$ if the signature is sensible).

Proof : Results from $Alg_{Exc}(SPEC) = Alg_{Lbl}(Tr(SPEC))$ and from $Gen_{Exc}(SPEC) = Gen_{Lbl}(Tr(SPEC))$ (or similarly from $Alg_{Exc}(\widehat{SPEC}) = Alg_{Lbl}(\widehat{Tr}(\widehat{SPEC}))$ and from $Gen_{Exc}(\widehat{SPEC}) = Gen_{Lbl}(\widehat{Tr}(\widehat{SPEC}))$ for constrained exception specifications) because $Tr(SPEC)$ is a positive conditional label specification. □

Definition 71 : An exception specification $SPEC$ is called *self complete* if and only if the algebra $\mathcal{T}_{SPEC}$ is self complete (see Definition 53).

Definition 72 : Let ΣExc_1 and ΣExc_2 be two exception signatures such that $\Sigma Exc_1 \subset \Sigma Exc_2$ (i.e. $S_1 \subset S_2$, $\Sigma_1 \subset \Sigma_2$ and $L_1 \subset L_2$). The forgetful functor U from $Alg_{Exc}(\Sigma Exc_2)$ to $Alg_{Exc}(\Sigma Exc_1)$ is defined as the forgetful functor on the underlying label algebras.

Remark 73 : This definition is sensible because for every exception algebra $\mathcal{A}$, the label algebra $U(\mathcal{A})$ is an exception algebra (i.e. satisfies the common future property). Indeed, from Theorem 48, U can also be shown as the forgetful functor from $Alg_{Lbl}(Fut_{\Sigma_2,L_2})$ to $Alg_{Lbl}(Fut_{\Sigma_1,L_1})$, the Fut_{Σ_i,L_i} being defined as in Theorem 48. (Consequently, this definition is also sensible for constrained exception signatures.)

Theorem 74 : Let $SPEC_1$ and $SPEC_2$ be two exception specifications such that $SPEC_1 \subset SPEC_2$. Let U be the forgetful functor from $Alg_{Lbl}(\Sigma Exc_2)$ to $Alg_{Lbl}(\Sigma Exc_1)$. The restriction of U to $Alg_{Lbl}(SPEC_2)$ can be co-restricted to $Alg_{Lbl}(SPEC_1)$.

More generally, given two exception signatures $\Sigma Exc_1 \subset \Sigma Exc_2$, for every ΣExc_2-algebra $\mathcal{A}$ and for every ΣExc_1-axiom φ (Ok or generalized) we have:

$$\mathcal{A} \models \varphi \implies U(\mathcal{A}) \models \varphi$$

Proof : If φ is a generalized axioms, it directly results from Theorem 27. If φ is an Ok-axiom, it results from Theorem 27 and from the fact that $\Sigma Exc_1 \subset \Sigma Exc_2$ implies $Tr_{\Sigma Exc_1}(\varphi) \subset Tr_{\Sigma Exc_2}(\varphi)$ (resp. $\widehat{Tr_{\Sigma Exc_1}}(\varphi) \subset \widehat{Tr_{\Sigma Exc_2}}(\varphi)$ for constrained algebras). □

Definition 75 : Let $SPEC_1$ and $SPEC_2$ be two exception specifications such that $SPEC_1 \subset SPEC_2$. The *synthesis functor* $F : Alg_{Exc}(SPEC_1) \to Alg_{Exc}(SPEC_2)$ is by definition the synthesis functor $F : Alg_{Lbl}(Tr(SPEC_1)) \to Alg_{Lbl}(Tr(SPEC_2))$ defined on label algebras in Theorem 32.

Remark 76 : If constrained exception specifications are involved, we have to consider the functor

$$F : Alg_{Lbl}(\widehat{Tr}(\widehat{SPEC}_1)) \to Alg_{Lbl}(\widehat{Tr}(\widehat{SPEC}_2)) \ .$$

Theorem 77 : Let $SPEC_1$ and $SPEC_2$ be two exception specifications such that $SPEC_1 \subset SPEC_2$. The synthesis functor $F : Alg_{Exc}(SPEC_1) \to Alg_{Exc}(SPEC_2)$ is a left adjoint for the forgetful functor $U : Alg_{Exc}(SPEC_2) \to Alg_{Exc}(SPEC_1)$.

Proof : Trivially results from $Alg_{Exc}(SPEC_i) = Alg_{Lbl}(Tr(SPEC_i))$ (as categories), from the definitions of U and F that coincide with the ones on label algebras, and from Theorem 33. (The two equalities $Alg_{Exc}(\widehat{SPEC}_i) = Alg_{Lbl}(\widehat{Tr}(\widehat{SPEC}_i))$ give the same result for the constrained case.) □

8.2 Structured exception specifications

The initiality results given in Section 8.1 can be used to define *hierarchical consistency* and *sufficient completeness* for structured exception specifications, in a similar way as in [GTW78].

Definition 78 : Let $SPEC_1$ and $SPEC_2$ be two exception specifications. The specification $SPEC_2$ is an *enrichment* of the specification $SPEC_1$ if and only if $SPEC_1 \subset SPEC_2$ (i.e. $S_1 \subset S_2$, $\Sigma_1 \subset \Sigma_2$, etc.). In this case, $SPEC_1$ is often called the *predefined* specification and $\Delta SPEC = SPEC_2 - SPEC_1$ is often called the *presentation of interest*.

Example 79 : *Stack* can be specified as an enrichment of *BoundedNat* (specified in Example 42).

ΔS : *Stack*

$\Delta\Sigma$:
$empty : \to Stack$
$push_\ _ : Stack\ Nat \to Stack$
$pop_ : Stack \to Stack$
$top_ : Stack \to Nat$

$$
\begin{array}{lll}
\Delta L & : & Underflow \\
 & & BadAccess \\
 & & \\
\Delta GenAx & : & empty \in Ok \\
 & & p \in Ok \ \wedge \ n \in Ok \implies (push\ p\ n) \in Ok \\
 & & pop\ empty \in Underflow \\
 & & top\ empty \in BadAccess \\
 & & \\
\Delta OkAx & : & top(push\ p\ n) = n \\
 & & pop(push\ p\ n) = p \\
\end{array}
$$

$$Where \ : \quad n \ : \ Nat \ ; \ \ p \ : \ Stack$$

This presentation is rather simple with respect to exception handling; no recovery is specified. Nevertheless, the most important thing is that the two Ok-axioms of $Stack$ only concern assignments σ such that both $\sigma(p)$ and $\sigma(n)$ are Ok-terms. In particular, the exceptional terms of the predefined specification *BoundedNat* are automatically excluded from the semantics of the Ok-axioms. For example, the term $top(push\ empty\ pred(0))$ does not result to $pred(0)$ (via the first Ok-axiom of $Stack$) because it is not an Ok-term.

Let us remember that hierarchical consistency means that the presentation does not introduce new identities over the predefined values (the so-called "no collapse" property). Without exception handling, hierarchical consistency it expressed by means of the unit of adjunction with respect to F and U: it must be injective on the initial object T_{SPEC_1}. With exception handling, the labels must also be taken into account. Hierarchical consistency must forbid the existence of new labelling of predefined terms by predefined labels. For example, *Stack* should not imply that $pred(0)$ becomes labelled by *TooLarge* if this is not a consequence of *BoundedNat*. A similar definition of hierarchical consistency has already been given in [BBC86].

Remark 80 : Let us remember that, from Theorem 77, for all $SPEC_1$-algebra $\mathcal{A}$, $Hom_{SPEC_1}(\mathcal{A}, U(F(\mathcal{A})))$ is canonically isomorphic to $Hom_{SPEC_2}(F(\mathcal{A}), F(\mathcal{A}))$ (see [McL71]). The *unit of adjunction* is the exception morphism from $\mathcal{A}$ to $U(F(\mathcal{A}))$ associated with the identity of $Hom_{SPEC_2}(F(\mathcal{A}), F(\mathcal{A}))$. This morphism is the morphism I_A defined in the proof of Theorem 33. If $\mathcal{A}$ is the initial exception algebra $\mathcal{T}_{SPEC_1}$, then $F(\mathcal{A}) = \mathcal{T}_{SPEC_2}$ because left adjoint functors preserve initial objects. Thus, the unit of adjunction $I_{\mathcal{T}_{SPEC_1}}$ is indeed the initial morphism from $\mathcal{T}_{SPEC_1}$ to $U(\mathcal{T}_{SPEC_2})$.

Definition 81 : Let $SPEC_1$ and $SPEC_2$ be two exception specifications such that $SPEC_1 \subset SPEC_2$. The associated enrichment is *hierarchically consistent* if and only if:

1. the unit of adjunction $I_{\mathcal{T}_{SPEC_1}}$ is injective;
2. $\forall l \in \widetilde{L_1}, \ \forall t \in T_{\Sigma_1}(T_{SPEC_1}), \ \ \overline{I_{\mathcal{T}_{SPEC_1}}}(t) \in l_{U(\mathcal{T}_{SPEC_2})} \implies t \in l_{\mathcal{T}_{SPEC_1}}$.

Remarks 82 :

1. The reverse implication of the second property is always satisfied since $I_{\mathcal{T}_{SPEC_1}}$ is an exception morphism ($\overline{I_{\mathcal{T}_{SPEC_1}}}$ preserves the labels).
2. The second property is also equivalent to

$$\forall l \in \widetilde{L_1},\ \ \overline{I_{\mathcal{T}_{SPEC_1}}}(l_{\mathcal{T}_{SPEC_1}}) = \overline{I_{\mathcal{T}_{SPEC_1}}}(T_{\Sigma_1}(\mathcal{T}_{SPEC_1})) \cap l_{U(\mathcal{T}_{SPEC_2})}$$

3. Equivalently, hierarchical consistency can be expressed in a more elegant way as "$I_{\mathcal{T}_{SPEC_1}}$ is partially retractable" (see [McL71] Chapter I.5 and [BBC86]).

The presentation *Stack* given in Example 79 is hierarchically consistent: no collapses are added in the sort Nat and no predefined labelling is added since the axioms only concern the new labels.

Let us remember that sufficient completeness means that the presentation does not add new values in the predefined sorts (the so-called "no junks" property). Without exception handling, sufficient completeness means that the unit of adjunction $I_{\mathcal{T}_{SPEC_1}}$ is surjective. With exception handling, such a definition is not suitable. For example, the term *top*(*empty*) is of predefined sort Nat but its value does not belong to the ones defined by $BoundedNat$. The value of *top*(*empty*) is a new error which has been introduced by the stack data structure, and there is no reason to take such errors into account when specifying $BoundedNat$. The only important point is that the presentation *Stack* allows us to deduce that *top*(*empty*) is erroneous (as it is labelled by $BadAccess$ and not recovered). Indeed, it is logical that, when specifying natural numbers, we do not foresee all the possible erroneous values introduced by all the possible ulterior enrichments. Thus, we have to accept new values of predefined sorts, provided that they are erroneous. A similar definition of hierarchical consistency has already been given in [BBC86].

Definition 83 : Let $SPEC_1$ and $SPEC_2$ be two exception specifications such that $SPEC_1 \subset SPEC_2$. The associated enrichment is *sufficiently complete* if and only if

$$U(\mathcal{T}_{SPEC_2} - \mathcal{T}_{SPEC_{2,Err}}) \ \subset\ I_{\mathcal{T}_{SPEC_1}}(\mathcal{T}_{SPEC_1})$$

(In the notation "$U(\mathcal{T}_{SPEC_2} - \mathcal{T}_{SPEC_{2,Err}})$," U should not be understood as the forgetful functor defined on exception algebras because $(\mathcal{T}_{SPEC_2} - \mathcal{T}_{SPEC_{2,Err}})$ is not an algebra. Here, U should be understood as the underlying forgetful functor simply defined on heterogeneous sets.)

The presentation *Stack* given in Example 79 is sufficiently complete: all the new values added to the sort Nat are erroneous, as they are futures of the terms *top*(*empty*) or *pop*(*empty*), which are labelled by $BadAccess$ or $Underflow$ respectively.

Remark 84 : In this section, we have more or less restricted our study to the initial approach. In particular, our definitions about "structured specifications" deal with the initial models only. This approach is not always satisfactory for specifying realistic

softwares, where a "loose" approach is often suitable. It has been shown in [Ber87] that it is not suitable to define hierarchical consistency or sufficient completeness on all the models of $Alg_{Exc}(SPEC)$ or $Gen_{Exc}(SPEC)$. More elaborated modular semantics must be used. However, even if we are interested in loose semantics only, initiality results are often very convenient to avoid the trivial models. For example, in [Bid89], the so called "basic specifications" (the ones that do not import other specifications) must have an initial model. Indeed, we believe that our definitions of hierarchical consistency and sufficient completeness provide a good starting point to define what "protecting an imported data structure" means in a loose approach. We have shown in particular that erroneous junks must be allowed; thus, a naive definition of "conservative extensions" is not convenient for exception or error handling. This fact should have an important impact on the general definitions of the semantics of modularity. It must be pointed out that all the existing frameworks for modularity ([AW86], [Bid89], [EBO91] and others), which follow a more or less "institution independent" viewpoint, are *not* suitable for a framework with exception or error handling because they all consider such a naive definition of conservativity (consequently, they do not allow erroneous junks such as $top(empty)$ for the stack module).

9 Some examples

Section 9.1 give several possible exception specifications of natural numbers. In order to give some insights into the semantics of exception specifications, the impact of the axioms on the initial algebra is described carefully.

Section 9.2 give several exception specifications of classical data structures. These specifications are not difficult to understand (as already mentioned, the specification of bounded natural number raises all the difficulties of exception handling for algebraic specifications); they mainly give an overview of how exception specifications look like.

9.1 Several versions for natural numbers

Let us first specify natural numbers without bound. The specification $Nat1$ given below, over the signature $NatExc = < \{Nat\}, \{0, succ_, pred_\}, \{Negative\} >$, is rather similar to the specifications of natural number that can be done in the framework of [GDLE84]: the two first generalized axioms mean that 0 and $succ$ are "safe operations."

$$GenAx \quad : \quad 0 \,\epsilon\, Ok$$
$$n \,\epsilon\, Ok \implies succ(n) \,\epsilon\, Ok$$

$$OkAx \quad : pred(succ(n)) = n$$

$$Where : \quad n : Nat$$

- The two first generalized axioms imply that the terms $succ^i(0)$ with $0 \leq i$ are Ok-terms.
- $pred(succ(n)) = n$ is an Ok-axiom; therefore it only applies to Ok-terms, in particular $n = pred(0)$ is not an acceptable assignment (to obtain the same result in the framework of [GDLE84] it is necessary to explicitly replace n by n_+, see Section 3.5).
- $pred(0)$ is not an Ok-term because the propagation of the label Ok through the Ok-axioms never matches the term $pred(0)$. More generally, the Ok-terms are the terms t such that every subterm of t contains at least as many occurrences of $succ$ than $pred$.
- The Ok-axiom implies that each Ok-term t is in the class of a term of the form $succ^i(0)$, where i is the difference between the number of occurrences of $succ$ and $pred$ in t.
- Notice that $pred(0)$ is not labelled by an exception name. Thus, in our framework, the specification $Nat1$ is not self complete. (The label $Negative$ is not used in $GenAx1$.)

All the examples $Nat2$ to $Nat7$ given in this section contain $Nat1$, and have the same signature $NatExc$. Moreover, $Nat2$ to $Nat7$ will not contain any additional Ok-axiom, nor additional generalized axiom with a conclusion of the form "$u \epsilon Ok$". Consequently, the set of Ok-terms will not change; $pred(0)$, even if recovered, will always remain exceptional. $Nat2$ to $Nat7$ show how the idea of labelling terms can be used in order to reach a very precise specification of fine exception handling features.

As $Nat1$ is not self complete, let us consider $Nat2$, which contains $Nat1$, such that $GenAx2$ contains $GenAx1$ and the following additional generalized axiom:

$$pred(0) \ \epsilon \ Negative$$

Since $pred(0)$ is labelled by $Negative$ and is not recovered, it is erroneous. More generally, $Nat2$ is self complete. In order to prove this fact, since T_{Nat2} is a quotient of the ground term algebra, and since there are no recoveries, it is sufficient to show that every ground term is either an Ok-term or in the class of a future of $pred(0)$. If t is neither Ok nor a future of $pred(0)$, then it is a future of $succ(0)$. Let $u = pred(succ^i(0))$ be the subterm of t defined by the innermost occurrence of $pred$ in t. u is necessarily an Ok-term; thus, it is in the class of the term $succ^{i-1}(0)$ (because the Ok-axiom applies). By induction, and since t contains more occurrences of $pred$ than $succ$, it comes that t is in the class of a future of $pred(0)$. This proves that $Nat2$ is self complete.

Let us remark that $Nat2$ implies to label only one term: $pred(0)$. There is no explicit propagation of the error raised by $pred$ on 0. The futures of $pred(0)$ are not labelled. For high quality software, it is sometimes convenient to require exception handlers that explicitly forward the exception names through the operations (if they do not recover them). Let us consider $Nat3$, which contains $Nat2$, such that $GenAx3$ contains $GenAx2$ and the following two additional generalized axioms:

$$n \,\epsilon\, Negative \implies succ(n) \,\epsilon\, Negative$$
$$n \,\epsilon\, Negative \implies pred(n) \,\epsilon\, Negative$$

The specification $Nat3$ clearly fulfills this requirement; all erroneous terms are labelled by $Negative$, reflecting the innermost error that has been raised in the term.

Let us note that, even if all the erroneous terms are labelled by $Negative$, they remain distinct. If we want to have a single erroneous value, as in the approach proposed in [GTW78], it is sufficient to consider $Nat4$, which contains $Nat3$, such that $GenAx4$ contains $GenAx3$ and the following additional generalized axiom:

$$n \,\epsilon\, Negative \implies n = pred(0)$$

$Nat4$ is an example where the generalized axioms are not only used for recovery purposes. Here, the additional axiom is used to collapse all the exceptional terms on a single erroneous value.

Instead of labelling the exceptional terms in order to reach self completeness (as in examples $Nat2$ to $Nat4$), it is possible to directly recover the non Ok-terms, without labelling them. Let us consider $Nat5$, which contains $Nat1$, such that $GenAx5$ contains $GenAx1$ and the following additional generalized axiom:

$$pred(0) = 0$$

This axiom is sufficient to recover all the exceptional ground terms. More precisely, $pred(0)$ is recovered on 0, $succ(pred(pred(0)))$ is recovered on $succ(0)$ and so on. Let us note that $Nat5$ remains nevertheless consistent because the terms $pred(0)$, $succ(pred(pred(0)))$, etc. are not Ok-terms, even if recovered; thus they are not acceptable assignments of the Ok-axioms. The initial model of $Nat5$ actually behaves on the Ok-part as natural numbers do.

In $Nat5$, $pred(0)$ is silently recovered, the intermediate exceptional state is not signaled (i.e. it is not labelled by an exception name). Even if recovered, this exception can be signaled by specifying $Nat6$, which contains $Nat5$, such that $GenAx6$ contains $GenAx5$ and the following additional generalized axiom:

$$pred(0) \,\epsilon\, Negative$$

Intuitively, we can consider that the label $Negative$ plays the role of a warning message.

Many other exception handling examples can be provided. For example the exception specification $Nat7$ over $NatExc = < \{Nat\}, \{0, succ_, pred_\}, \{Negative\} >$ differs from the two previous recovery cases:

$$
\begin{array}{lll}
GenAx & : & 0 \,\epsilon\, Ok \\
 & & n \,\epsilon\, Ok \implies succ(n) \,\epsilon\, Ok \\
 & & pred(0) \,\epsilon\, Negative \\
 & & n \,\epsilon\, Negative \implies pred(n) \,\epsilon\, Negative \\
 & & n \,\epsilon\, Negative \wedge n = m \implies m \,\epsilon\, Negative \\
 & & succ(pred(n)) = n \\
 & & \\
OkAx & : & pred(succ(n)) = n
\end{array}
$$

$Nat7$ does not directly describe an explicit recovery of a particular exceptional term (as the axiom $pred(0) = 0$ does). It describes a general property, $succ(pred(n)) = n$, that concerns exceptional cases as well as normal cases. Several instancies are in fact recoveries, providing that the exceptional term under consideration contains more occurrence of *succ* than *pred*. For instance, $succ(succ(pred(0)))$ is recovered on the Ok-term $succ(0)$. This example is similar to the example of "recovery by rearrangement of algebraic expressions" given in Section 3.7.

We sum up in figure 9.1 the different features of $Nat1$ to $Nat7$.

3— Different features on the specifications of natural numbers

	$Nat1$	$Nat2$	$Nat3$	$Nat4$	$Nat5$	$Nat6$	$Nat7$
recovery	No	No	No	No	Yes	Yes	Yes
labelling	No	Yes	Yes	Yes	No	Yes	Yes
explicit propagation of labelling	No	No	Yes	Yes	No	No	Yes
property on exceptional terms	No	No	No	Yes	No	No	Yes

9.2 Other examples

We give an example of exception specification of binary trees over natural numbers.

ΔS : $Tree$

$\Delta\Sigma$: $empty :\to Tree$
$node_\ _\ _ : Tree\ Nat\ Tree \to Tree$
$root_ : Tree \to Nat$
$left_ : Tree \to Tree$
$right_ : Tree \to Tree$

ΔL : $Void$

$\Delta GenAx$: $empty \in Ok$
$t1 \in Ok \wedge t2 \in Ok \wedge n \in Ok \implies node(t1, n, t2) \in Ok$
$root(empty) \in Void$
$left(empty) \in Void$
$right(empty) \in Void$
$t \in Void \implies t = empty$

$\Delta OkAx$: $left(node(t1, n, t2)) = t1$
$right(node(t1, n, t2)) = t2$
$root(node(t1, n, t2)) = n$

$$Where\ :\quad n\ :\ Nat\ ;\quad t, t1, t2\ :\ Tree$$

$BinTree$ is a presentation of interest, whose predefined specification can be any specification of natural numbers (e.g. $Nat2$ to $Nat7$). Let us note that the exception label $Void$ intersect two different sorts ($Tree$ and Nat) because for example the terms $left(empty)$ and $root(empty)$ are labelled by $Void$. In this example, we have chosen to recover every exceptional binary tree labelled by $Void$ on the empty tree. Let us note that the variable t in the last generalized axiom is of sort $Tree$; thus, the exceptional term $root(empty)$, of sort Nat, is of course not collapsed with the empty tree because it is not an acceptable assignment of t.

We give an example of exception specification of queues (FIFO) over the natural numbers.

ΔS : $Queue$

$\Delta\Sigma$:
$$\begin{aligned}
&empty\ :\ \to Queue\\
&add_\ _\ :\ Nat\ Queue \to Queue\\
&remove_\ :\ Queue \to Queue\\
&first_\ :\ Queue \to Nat
\end{aligned}$$

ΔL : $Underflow\ ,\ ErrorQueue\ ,\ ErrorFirst$

$\Delta GenAx$:
$$\begin{aligned}
&empty \in Ok\\
&q \in Ok\ \wedge\ n \in Ok \implies add(n,q) \in Ok\\
&remove(empty) \in Underflow\\
&first(empty) \in ErrorFirst\\
&q \in Underflow \implies remove(q) \in ErrorQueue\\
&q \in Underflow\ \wedge\ n \in Ok \implies add(n,q) = add(n, empty)
\end{aligned}$$

$\Delta OkAx$:
$$\begin{aligned}
&remove(add(n, empty)) = empty\\
&remove(add(n,(add(m,q)))) = add(n,(remove(add(m,q))))\\
&first(add(n, empty)) = n\\
&first(add(n, add(m,q)))) = first(add(m,q))
\end{aligned}$$

$$Where\ :\quad n, m\ :\ Nat\ ;\quad q\ :\ Queue$$

Let us note that we have specified two kind of exceptional queues. On the one hand, if only one *remove* is applied to the empty queue (leading to $Underflow$), then it remain possible to *add* an *Ok* natural number to it; the resulting queue is recovered to the queue containing this natural number. (Notice that, however, a queue labelled by $Underflow$ is not directly recovered to the empty queue.) On the other hand, if another *remove* is applied to such an erroneous queue, or if an erroneous natural number is added to a queue, then there are no specified recoveries; only one exceptional application of *remove* is allowed for recovering *add*. This exception specification gives a good example where partial

functions are not powerful enough to describe the same data structure; $remove(empty)$ is not defined, but $add(n, remove(empty))$ is defined (see Section 3.3).

We give an example of bounded stacks over the natural numbers. It is rather similar to the bounded natural number example studied so far in the paper.

ΔS : $Stack$

$\Delta\Sigma$:
$empty\ :\ \rightarrow Stack$
$push_\ _\ :\ Nat\ Stack \rightarrow Stack$
$pop_\ :\ Stack \rightarrow Stack$
$top_\ :\ Stack \rightarrow Nat$

ΔL : $Underflow\ ,\ Overflow\ ,\ ErrorTop$

$\Delta GenAx$:
$\bigwedge_{i=1..Max} x_i \in Ok \implies push(x_1, push(x_2, \ldots, push(x_{Max}, empty))) \in Ok$
$push(x, X) \in Ok \implies X \in Ok$
$pop(empty) \in Underflow$
$top(empty) \in ErrorTop$
$push(x_1, push(x_2, \ldots, push(x_{Max+1}, empty))) \in Overflow$
$push(x, X) \in Overflow \implies push(x, X) = X$

$\Delta OkAx$:
$pop(push(x, X)) = X$
$top(push(x, X)) = X$

$Where\ :\ x_1, \ldots, x_{Max+1},\ x\ :\ Nat\ ;\ X\ :\ Stack$

The last generalized axiom means "if the operation *push* raises $Overflow$ then do not perform it." Let us remember that this specification is consistent within our framework; it would lead to a trivial initial algebra in all the other existing frameworks because it requires for the exception label $Overflow$ to be carried by terms, not by values (see Example 2).

In practice, it is generally rather unpleasant to deal with terms such as the term $push(x_1, \ldots, push(x_{Max}, empty))$. We would prefer to specify the height of a stack and use it to characterize the Ok-terms. This lead to the following specification:

ΔS : $Stack$

$\Delta\Sigma$:
$empty\ :\ \rightarrow Stack$
$push_\ _\ :\ Nat\ Stack \rightarrow Stack$
$height_\ :\ Stack \rightarrow Nat$
$pop_\ :\ Stack \rightarrow Stack$
$top_\ :\ Stack \rightarrow Nat$

ΔL : $Underflow\ ,\ Overflow\ ,\ ErrorTop$

$$
\begin{array}{lll}
\Delta GenAx & : & empty \,\epsilon\, Ok \\
& & X \,\epsilon\, Ok \;\wedge\; height(X) < Max = true \;\wedge\; x \,\epsilon\, Ok \implies push(x,X) \,\epsilon\, Ok \\
& & pop(empty) \,\epsilon\, Underflow \\
& & top(empty) \,\epsilon\, ErrorTop \\
& & height(X) = Max \implies push(x,X) \,\epsilon\, Overflow \\
& & push(x,X) \,\epsilon\, Overflow \implies push(x,X) = X
\end{array}
$$

$$
\begin{array}{lll}
\Delta OkAx & : & height(empty) = 0 \\
& & height(push(x,X)) = succ(height(X)) \\
& & pop(push(x,X)) = X \\
& & top(push(x,X)) = X
\end{array}
$$

$$
Where : \quad x : Nat \;;\quad X : Stack
$$

Notice that the before last generalized axiom *cannot* be replaced by

$$
Max < height(X) = true \implies X \,\epsilon\, Overflow
$$

because the operation *height* is only defined on Ok-terms (as *height* is defined in $OkAx$). Moreover, putting the axioms defining *height* in $GenAx$ without adding any precondition is not hierarchically consistent because it would lead to $succ(Max) = Max$, according to the last recovery axiom. Similarly, the two first generalized axioms *cannot* be replaced by

$$
height(X) \leq Max = true \implies X \,\epsilon\, Ok
$$

because it does not imply that the Ok-stacks only contain Ok natural numbers; moreover, in this case, there would be no Ok-term of sort stack at all in the initial algebra if *height* remains defined in $OkAx$.

An example of exception specification of intervals is given below, where the interval $[3,8]$ is specified (with $\Sigma Exc =< \{Interv\}, \{0, succ_, pred_\}, \{TooLow, TooLarge\} >$).

$$
\begin{array}{lll}
GenAx & : & succ^8(0) \,\epsilon\, Ok \\
& & succ^4(n) \,\epsilon\, Ok \implies succ^3(n) \,\epsilon\, Ok \\
& & succ^2(0) \,\epsilon\, TooLow \\
& & succ(n) \,\epsilon\, TooLow \implies n \,\epsilon\, TooLow \\
& & pred(succ^3(0)) \,\epsilon\, TooLow \\
& & succ^9(0) \,\epsilon\, TooLarge
\end{array}
$$

$$
\begin{array}{lll}
OkAx & : & pred(succ(n)) = n
\end{array}
$$

$$
Where : \quad n : Interv
$$

The only interesting point of this example is to illustrate the fact that a subterm of an Ok-term is not necessarily an Ok-term ($succ^3(0)$ is Ok while $succ^2(0)$ is not).

Our last example belong to the "dynamic" class of exceptional cases. We give a specification of (bounded) arrays, where a new array is not supposed initialized. The ranges

of an array are of sort $Index$, that can be any sort such that the boolean operations "$<$" "$\leq$" and "eq" are provided; usually it is required for "$\leq$" to define a total order; natural numbers can be used for example. The elements stored in the array belong to the sort $Elem$, which can be any sort.

ΔS : $Array$

$\Delta\Sigma$:
$create_\ _ \ : \ Index\ Index \rightarrow Array$
$store_\ _\ _ \ : \ Elem\ Array\ Index \rightarrow Array$
$fetch_\ _ \ : \ Array\ Index \rightarrow Elem$
$lower_ \ : \ Array \rightarrow Index$
$upper_ \ : \ Array \rightarrow Index$

ΔL : $BadRange$, $OutOfRange$, $NonInitialized$

$\Delta GenAx$:

$$low \in Ok \ \wedge up \in Ok \ \wedge \ low < up = true \implies create(low, up) \in Ok$$

$$\left\{ \begin{array}{l} a \in Ok \ \wedge \ ind \in Ok \ \wedge \ x \in Ok \\ \qquad \wedge \ lower(a) \leq ind = true \\ \qquad \wedge \ ind \leq upper(a) = true \end{array} \right\} \Rightarrow store(x, a, ind) \in Ok$$

$$low < up = false \implies create(low, up) \in BadRange$$

$$ind < lower(a) = true \implies store(x, a, ind) \in OutOfRange$$

$$upper(a) < ind = true \implies store(x, a, ind) \in OutOfRange$$

$$ind < lower(a) = true \implies fetch(a, ind) \in OutOfRange$$

$$upper(a) < ind = true \implies fetch(a, ind) \in OutOfRange$$

$$lower(a) \leq ind = true \ \wedge \ ind \leq upper(a) = true \implies fetch(create(low, up), ind)) \in NonInitialized$$

$$eq(ind1, ind2) = false \ \wedge \ fetch(a, ind1) \in NonInitialized \implies fetch(store(x, a, ind2), ind1) \in NonInitialized$$

$\Delta OkAx$:

$$lower(create(low, up)) = low$$

$$upper(create(low, up)) = up$$

$$lower(store(x, a, ind)) = lower(a)$$

$$upper(store(x, a, ind)) = upper(a)$$

$$store(x, store(y, a, ind), ind) = store(x, a, ind)$$

$$eq(ind1, ind2) = false \implies store(x, store(y, a, ind1), ind2) = store(y, store(x, a, ind2), ind1)$$

$$fetch(store(x, a, ind), ind) = x$$

$Where$: $low,\ up,\ ind,\ ind1,\ ind2 \ : \ Index \ ; \ \ x,\ y \ : \ Elem \ ; \ \ a \ : \ Array$

The term $create(low, up)$ creates a new array of range $[low, up]$. The operations $lower$ and $upper$ retrieve the acceptable range of an array. Notice that the last generalized axiom is useful, even if it seems redundant with the three last Ok-axioms, because the Ok-axioms only concern the Ok-terms, while the purpose of the last generalized axiom is to label erroneous terms. Another possibility would be to remove the last generalized axiom and to move the three last Ok-axioms into $GenAx$ (then, they would apply to all terms, including the exceptional ones).

10 Conclusion

We have shown that exception handling requires a refined notion of the satisfaction relation for algebraic specifications. The scope of an axiom must be restricted to carefully chosen patterns, because a satisfaction relation based on assignments with range in *values* often raises inconsistencies. A more elaborated notion of assignment must be considered: assignment with range in *terms*. This allows us to restrict the scope of an axiom to certain suitable patterns, and solves the inconsistencies raised by exception handling.

We have also shown that exception names, or error messages, must be carried by terms, and that they are advantageously reflected by *labels*. Labels must not go through equational atoms; thus, two terms having the same value do not necessarily carry the same labels. We have first defined the framework of *label algebras*, that defines suitable semantics for labels. The scope of the label axioms is carefully delimited by labels which serve as special marks on terms.

Then, we have proposed a new algebraic framework for exception handling, based on label algebras, which is powerful enough to cope with all suitable exception handling features such as implicit propagation of exceptions, possible recoveries, declaration of exception names, etc. As shown in Section 9, all the exceptional cases can easily be specified ("intrinsic" exceptions of an abstract data type, "dynamic" exceptional cases and bounded data structures). This approach solves all the inconsistencies raised by all the existing frameworks (see Section 3) and succeeds with respect to *clarity* and *terseness* of specifications, that are two crucial criteria for formal specifications with exception handling. More precisely, clarity and terseness are obtained because two different kinds of axioms have been distinguished, with distinct implicit semantics: the generalized axioms treat the exceptional cases, and the *Ok*-axioms only treat the normal cases.

The usual inconsistencies raised by exception handling for algebraic specifications are solved in our framework because we carefully define the difference between *exception* and *error*. An error is an exception which has not been recovered. Even if an exceptional term has been recovered, it remains exceptional because an exceptional treatment has been required in its "history."

Although we have introduced the theory of label algebras as a general frame for exception handling purpose, the application domain of label algebras seems to be much more general than exception handling. Indeed, labels provide a great tool to express several other features developed in the field of (first order) algebraic specifications. We have outlined in Section 4.2 how label algebras can be used to specify several more standard algebraic approaches such as order sorted algebras [Gog83], partial functions [BW82] or observability issues [Hen89][BB91]. However, all the specific applications of label algebras require certain *implicit* label axioms in order to preserve clarity and terseness. Thus, the framework of label algebras provides us with "low level" algebraic specifications: in a generic way, the specific semantical aspects of a given approach (e.g. subsorting or exception handling) can be specified by a well chosen set of label axioms.

Although we have studied "structured" exception specifications in Section 8.2, we have not studied "modular constraints" according to elaborated modular semantics such as the

ones of [AW86], [Bid89] or [EBO91]. Nevertheless, we have shown in Section 5 that the framework of label algebras restricted to positive conditional axioms, and consequently the one of exception algebras, form a specification logic which has free constructions [EBO91][EBCO91].[8] These results provide us with a first basis to study more elaborated notions of modularity for label specifications. However, modularity should indeed be studied according to the specific application under consideration (behavioural specifications, exception specifications, etc.): for instance, we have shown in Section 8.2 that the definition of "sufficient completeness" for exception specifications must allow erroneous junks. We have also pointed out in Section 8.2 that the existing frameworks for modularity do not cope with exception handling because they do not allow erroneous junks. Consequently, the definition of a suitable modular approach capable of treating algebraic frameworks with exception handling remain an open question.

Several other extensions of the framework of label algebras will probably give promising results. Intuitively, labels are unary predicates on terms. In order to facilitate certain applications of label algebras, we plane to generalize labels to "labels of strictly positive arity." Theorem proving methods according to the semantics of label algebras should be studied in future works (in the spirit of the proof of completeness given in Section 9.1). Higher order label specifications may also be dealt with in future works.

Last, but not least, let us mention that bounded data structures play a crucial role in the theory of *testing* because test data sets should contain many elementary tests near the bounds. One of our current researches is to extend the theory of test data selection from algebraic specifications described in [BGM91] to exception specifications.

Acknowledgements: We would like to thank Pierre Dauchy and Anne Deo-Blanchard for a careful reading of the draft version of this paper. Several discussions with Marie-Claude Gaudel and Michel Bidoit have been helpful. This work has been partially supported by CNRS GRECO de Programmation and EEC Working Group COMPASS.

References

[AC91] Astesiano E., Cerioli M. : "*Non-strict don't care algebras and specifications*" TAPSOFT CAAP, Brighton U.K., April 1991, Springer-Verlag LNCS 493, p.121-142.

[AW86] Astesiano E., Wirsing M. : "*An introduction to ASL*" Proc. of the IFIP WG2.1 Working Conference on Program Specifications and Transformations, 1986.

[BB91] Bernot G., Bidoit M. : "*Proving the correctness of algebraically specified software: Modularity and Observability issues*" Proc. of AMAST-2, Second Conference of Algebraic Methodology and Software Technology, Iowa City, Iowa, USA, May 1991.

[BBC86] Bernot G., Bidoit M., Choppy C. : "*Abstract data types with exception handling : an initial approach based on a distinction between exceptions and errors*"

[8] but it does not form a liberal institution [GB84].

Theoretical Computer Science, Vol.46, n.1, pp.13-45, Elsevier Science Pub. B.V. (North-Holland), November 1986. (Also LRI Report 251, Orsay, Dec. 1985.)

[BBK91] Bernot G., Bidoit M., Knapik T. : "*Observational approaches in algebraic specifications: A comparative study*" LIENS report 91-6, Ecole Normale Superieure, Paris, April 1991.

[Ber86] Bernot G. : "*Une sémantique algébrique pour une spécification différenciée des exceptions et des erreurs : application à l'implémentation et aux primitives de structuration des spécifications formelles*" Thèse de troisième cycle, Université de Paris-Sud, Orsay, February 1986.

[Ber87] Bernot G. : "*Good functors ... are those preserving philosophy !*" Proc. Summer Conference on Category Theory and Computer Science, September 1987, Springer-Verlag LNCS 283, pp. 182-195

[BGM91] Bernot G., Gaudel M.C., Marre B. : "*Software testing based on formal specifications: a theory and a tool*" to appear in Software Engineering Journal, December 1991.

[Bid84] Bidoit M. : "*Algebraic specification of exception handling by means of declarations and equations*" Proc. 11th ICALP, Springer-Verlag LNCS 172, July 1984.

[Bid89] Bidoit M. : "*Pluss, un langage pour le développement de spécifications algébriques modulaires*" Thèse d'état, University of Paris-Sud, 1989.

[BW82] Broy M., Wirsing M. : "*Partial abstract data types*" Acta Informatica, Vol.18-1, Nov. 1982.

[Com90] Comon H. : "*Equational formulas in order-sorted algebras*" Proc. ICALP, Warwick, Springer-Verlag, July 1990.

[EBCO91] Ehrig H., Baldamus M., Cornelius F., Orejas F. : "*Theory of algebraic module specification including behavioural semantics, constraints and aspects of generalized morphisms*" Proc. of AMAST-2, Second Conference of Algebraic Methodology and Software Technology, Iowa City, Iowa, USA, May 1991.

[EBO91] Ehrig H., Baldamus M., Orejas F. : "*New concepts for amalgamation and extension in the framework of specification logics*" Research report Bericht-No 91/05, Technische Universitat Berlin, May 1991.

[EM85] Ehrig H., Mahr B. : "*Fundamentals of Algebraic Specification 1. Equations and initial semantics*" EATCS Monographs on Theoretical Computer Science, Vol.6, Springer-Verlag, 1985.

[FGJM85] Futatsugi K., Goguen J., Jouannaud J-P., Meseguer J. : "*Principles of OBJ2*" Proc. 12th ACM Symp. on Principle of Programming Languages, New Orleans, january 1985.

[GB84] Goguen J.A., Burstall R.M. : "*Introducing institutions*" Proc. of the Workshop on Logics of Programming, Springer-Verlag LNCS 164, pp.221-256, 1984.

[GDLE84] Gogolla M., Drosten K., Lipeck U., Ehrich H.D. : "*Algebraic and operational semantics of specifications allowing exceptions and errors*" Theoretical Computer Science 34, North Holland, 1984, pp.289-313.

[GM89] Goguen J.A., Meseguer J. : "*Order-sorted algebra I: equational deduction for multiple inheritance, overloading, exceptions and partial operations*" Technical Report SRI-CSL-89-10, SRI, July 1989.

[Gog78a] Goguen J.A. : "*Abstract errors for abstract data types*" Formal Description of Programming Concepts, E.J. NEUHOLD Ed., North Holland, pp.491-522, 1978.

[Gog78b] Goguen J.A. : "*Order sorted algebras: exceptions and error sorts, coercion and overloading operators*" Univ. California Los Angeles, Semantics Theory of Computation Report n.14, Dec. 1978.

[Gog83] Gogolla M. : "*Algebraic specification with partially ordered sorts and declarations*" Research report Forschungsbericht No 169, University of Dortmund, 1983.

[Gog84] Gogolla M. : "*Partially ordered sorts in algebraic specifications*" Proc. 9th Colloquium on Trees in Algebra and Programming (CAAP), Bordeaux, Bruno Courcelle (Ed.), Cambridge University Press, Cambridge (1984), pp. 139-153.

[Gog87] Gogolla M. : "*On parametric algebraic specifications with clean error handling*" Proc. Joint Conf. on Theory and Practice of Software Development, Pisa (1987), Springer-Verlag LNCS 249, pp.81-95.

[GTW78] Goguen J.A., Thatcher J.W., Wagner E.G. : "*An Initial Algebra Approach to the Specification, Correctness, and Implementation of Abstract Data Types*" Current Trends in Programming Methodology, ed. R.T. Yeh, Printice-Hall, Vol.IV, pp.80-149, 1978. (Also IBM Report RC 6487, October 1976.)

[Gut75] Guttag J.V. : "*The specification and application to programming*" Ph.D. Thesis, University of Toronto, 1975

[Hen89] Hennicker R. : "*Implementation of Parameterized Observational Specifications*" TapSoft, Barcelona, LNCS 351, vol.1, pp.290-305, 1989.

[LG86] Liskov B., Guttag J. : "*Abstraction and specification in program development*" The MIT Press and McGraw-Hill Book Company, 1986.

[LZ75] Liskov B., Zilles S. : "*Specification techniques for data abstractions*" IEEE Transactions on Software Engineering, Vol.SE-1 n.1, March 1975.

[McL71] Mac Lane S. : "*Categories for the working mathematician*" Graduate texts in mathematics, 5, Springer-Verlag, 1971

[Pla82] Plaisted D. : "*An initial algebra semantics for error presentations*" Unpublished Draft, 1982

[Sch91] Schobbens P.Y. : "*Clean algebraic exceptions with implicit propagation*" Proc. of AMAST-2, Second Conference of Algebraic Methodology and Software Technology, Iowa City, Iowa, USA, May 1991.

[Smo86] Smolka G. : "*Order-sorted horn logic: semantics and deduction*" Research report SR-86-17, Univ. Kaiserslautern, Oct. 1986.

[WB80] Wirsing M., Broy M. : "*Abstract data types as lattices of finitely generated models*" Proc. of the 9th Int. Symposium on Mathematical Foundations of Computer Science (MFCS), Rydzyna, Poland, Sept. 1980.

Chapitre 7 :
Good functors ... are those preserving philosophy !

Gilles BERNOT

LIENS, CNRS URA 1327
Ecole Normale Supérieure,
45 Rue d'Ulm,
F-75230 PARIS Cédex 05,
FRANCE

bitnet: bernot@frulm63
uucp: bernot@ens.ens.fr

(Appeared in Proc. of "Category Theory and Computer Science", LNCS 283, 1987.)

Contents

1 Introduction **229**

2 Elementary reminders **229**

3 Forgetful and synthesis functors **231**

4 Consistency and completeness **232**

4.1 Sufficient completeness 232

4.2 Hierarchical consistency 233

5 Combining presentations **234**

5.1 Sufficient completeness 234

5.2 Hierarchical consistency 235

6 Loose semantics with "Protect" **236**

7 Conclusion **238**

Résumé

Le but de cet article est de mettre en garde le chercheur en types abstraits algébriques contre une utilisation abusive de la théorie des catégories. Quelques propriétés peu souhaitables du (pourtant classique) *foncteur de synthèse* sont décrites, spécialement si l'on s'intéresse aux sémantiques dites “loose”. Tous les résultats énoncés ici sont particulièrement simples, sinon triviaux ; néanmoins, ils illustrent des faits donnant lieu à de nombreuses erreurs dans le cadre des types abstraits algébriques. Ces erreurs résultent souvent d'une inadéquation entre certains outils catégoriques bien connus et le concept informatique que l'on souhaite modéliser. Enfin, une approche hiérarchique fondée sur la catégorie des modèles “protégeant les sortes prédéfinies” est proposée, et les premières propriétés en sont dégagées.

Mots clés : complétude, consistance, modèle initial, spécifications abstraites, spécifications structurées, théorie des catégories, types abstraits algébriques.

Abstract

The aim of this paper is to prevent the abstract data type researcher from an improper, naive use of category theory. We mainly emphasize some unpleasant properties of the *synthesis functor* when dealing with so-called *loose semantics* in a hierarchical approach. All our results and counter-examples are very simple, nevertheless they shed light on many common errors in the abstract specification field.
We also summarize some properties of the category of models “protecting predefined sorts.”

Keywords : abstract data types, abstract specifications, category theory, completeness, consistency, initial model, structured specifications.

1 Introduction

In the following pages, we focus our attention on results which seem to be "trivially ensured" in the basic abstract data type framework. We sometimes give proofs... often counter-examples of such results. In order to get striking counter-examples, we provide very simple ones, if not trivial (mainly based on elementary algebraic properties of natural numbers). Nevertheless, many common errors, or misinterpretations found in the abstract data type litterature result from similar mechanisms. This emphasizes the fact that category theory should be carefully used in the abstract data type field, including for (very) low level concepts.
More provocatively: this paper mainly points out the fact that the synthesis functor $\mathcal{F}$ of abstract data types "does not preserve philosophy." However, since about teen years [ADJ76], it is well known that this functor is crucial for defining a hierarchical, modular approach of abstract specifications!

Some elementary reminders about abstract data types are given in the next section (Section 2). Section 3 discusses about the well known *forgetful* and *synthesis functors*, $\mathcal{U}$ and $\mathcal{F}$, associated with a hierarchical approach. Section 4 shows the difficulty of properly defining sufficient completeness and hierarchical consistency with loose semantics. In Section 5, we show what happens when combining enrichments. Lastly, Section 6 discusses about a loose semantics obtained by "protecting" predefined sorts.

The following discussions are mainly centered on pairs [positive fact / proof] (respectively: [negative fact / counter-example]).

2 Elementary reminders

Let us begin with basic definitions and properties [ADJ76]:
Given a *signature* Σ (i.e. a finite set S of *sorts* and a finite set Σ of *operation-names* with arity in S), a Σ-algebra, A, is a heterogeneous set partitioned as $\{ A_s \}_{s \in S}$, and for each operation-name $op : s_1 \cdots s_{n-1} \to s_n$ of Σ there is an operation $op_A : A_{s_1} \times \cdots \times A_{s_{n-1}} \to A_{s_n}$.
A Σ-*morphism* from A to B is a sort-preserving, operation-preserving application from A to B. This defines a category, denoted by $Alg(\Sigma)$; it has an initial object: the ground-term algebra T_Σ .
In the following, a *specification SPEC* will be defined by a signature Σ and a finite set E of *positive conditional equations* of the form:

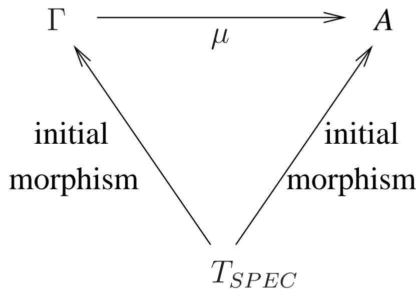
$$v_1 = w_1 \ \wedge \cdots \wedge \ v_{n-1} = w_{n-1} \Longrightarrow v_n = w_n$$

where v_i and w_i are Σ-terms with variables.
Given a specification *SPEC*, $Alg(SPEC)$ is the full sub-category of $Alg(\Sigma)$ whose objects are the Σ-algebras which validate each axiom of E. The category $Alg(SPEC)$ has an initial object, denoted by T_{SPEC} [BPW82].
Since T_{SPEC} exists, $Gen(SPEC)$ can be defined as the full sub-category of $Alg(SPEC)$ such that the initial morphism is an epimorphism (i.e. is *surjective*, in our framework). $Gen(SPEC)$ is the category of the *finitely generated algebras*. Our first "fact" will be devoted to the following remark:
It is well known that $Gen(SPEC)$ is a particularly interesting category for the abstract

data type computer scientist; nevertheless, this is not exactly due to its large spectrum of morphisms, as reminded below.

fact 1 : (Morphisms from a finitely generated algebra)
Let Γ be an object of $Gen(SPEC)$ and A an object of $Alg(SPEC)$. The set $Hom_{Alg(SPEC)}(\Gamma, A)$ contains *at most one element*. Consequently, for all objects X and Y of $Gen(SPEC)$, $Hom_{Gen(SPEC)}(X, Y)$ contains at most one morphism.

Proof: By initiality properties, if there exists a morphism μ, then the following triangle commutes:



Thus, the unicity of μ results from the surjectivity of the initial morphism associated with Γ. □

One of the most important aspect of abstract data types is its structured, hierarchical, modular approach. This is obtained by means of *presentations*. A presentation $PRES$ over $SPEC$ is a new "part of specification" $PRES = < S', \Sigma', E' >$ such that the disjoint union $SPEC + PRES = < S \cup S',\ \Sigma \cup \Sigma',\ E \cup E' >$ is a specification. Sorts and operations of $SPEC$ are often called the *predefined* sorts and operations. Relations between the categories $Alg(SPEC)$ and $Alg(SPEC + PRES)$ are handled by the well known *forgetful functor* and *synthesis functor*:

$$(\mathcal{U} \ : \ Alg(SPEC + PRES) \rightarrow Alg(SPEC)) \text{ and}$$
$$(\mathcal{F} \ : \ Alg(SPEC) \rightarrow Alg(SPEC + PRES)).$$

The functor $\mathcal{F}$ is a left adjoint for the functor $\mathcal{U}$. Consequently, for each $SPEC$-algebra A, there is a particular morphism from A to $\mathcal{U}(\mathcal{F}(A))$: the morphism deduced from the adjunction unit (or *adjunction morphism*). This morphism is absolutely crucial for the hierarchical approach: it allows to evaluate the modifications performed on A under the action of $PRES$.

Example 2 : If A is equal to $\mathbb{N}$ over the signature $\{0,\ succ_\}$ (without axioms) and if $PRES$ adds $pred_$ with the axioms $[pred(succ(n)) = succ(pred(n)) = n]$, then $\mathcal{U}(\mathcal{F}(\mathbb{N}))$ is isomorphic to $\mathbb{Z}$. The unit of adjunction leads to the natural inclusion; and this morphism permits to show that $\mathbb{N}$ has been modified by adding negative values.
If the axioms were $[pred(succ(n)) = n$ and $pred(0) = 0]$, then the unit of adjunction leads to the identity over $\mathbb{N}$ showing that this second specification of $pred$ does not change $\mathbb{N}$.

3 Forgetful and synthesis functors

We first present a rather obvious reminder about the *forgetful functor*. Let B be a $SPEC+PRES$-algebra. The forgetful functor removes all subsets B_s where $s \in S'$, and all operations of Σ' are forgotten (including those with arity in S only), but *it does not remove any value of predefined sort*: $\mathcal{U}(B_s) = B_s$ for each $s \in S$. For instance, in Example 2, $\mathcal{U}(\mathbb{Z}) = \mathbb{Z} \neq \mathbb{N}$.

Let us remind the classical definition of the *synthesis functor* (although classical, this definition is the starting point of some misinterpretations!): Let A be a $SPEC$-algebra and let $T_{\Sigma+\Sigma'}(A)$ be the algebra of $\Sigma + \Sigma'$-terms with variables in A; we denote by $eval : \mathcal{U}(T_{\Sigma+\Sigma'}(A)) \rightarrow A$ the canonical evaluation morphism. $\mathcal{F}(A)$ is the quotient of $T_{\Sigma+\Sigma'}(A)$ by the smallest congruence containing both the fibers of *eval* and the close instanciations of $E + E'$.
Because $E + E'$ is required in the definition of $\mathcal{F}$ (instead of E' alone), $\mathcal{F}(A)$ does not only depend on A and $PRES$; it also depends on $SPEC$.

fact 3 : Given a presentation $PRES$, the action of the synthesis functor $\mathcal{F}$ over a given, fixed algebra A is highly dependent of the predefined specification.
As outlined in the following example, this fact considerably restricts the possibility of writing "implementation independent" specifications (see for instance [EKMP80], [SW82], or [BBC86a] about abstract implementations).

Example 4 : Let $SPEC$ be a classical specification of NAT with operations 0, $succ_$ and $_+_$:

$$x + 0 = x$$
$$x + succ(y) = succ(x + y)$$

Let $SPEC'$ be the specification obtained by adding the following axiom to $SPEC$:

$$x + y = x + z \Longrightarrow y = z$$

The specifications $SPEC$ and $SPEC'$ have clearly the same initial object: $\mathbb{N}$.
Let $PRES$ be the presentation adding no sort, adding the operation $_ \times _$, and adding the axioms:

$$x \times succ(0) = x > \text{(1 is neutral)}$$
$$x \times succ(y) = x + (x \times y) > \text{(recursive definition)}$$

When $PRES$ is shown as a presentation over $SPEC$, $\mathcal{F}(\mathbb{N})$ is a model where all terms containing a multiplication by 0 cannot be evaluated. When $PRES$ is shown as a presentation over $SPEC'$, $\mathcal{F}(\mathbb{N})$ is isomorphic to $\mathbb{N}$, because:

$$x + 0 = x = x \times succ(0) = x + (x \times 0)$$

and the simplification axiom of $SPEC'$ leads to $0 = x \times 0$.
Notice that, in spite of the fact that $SPEC$ and $SPEC'$ have the same initial semantics, the presentation $PRES$ is not completely specified over the first specification, but is completely specified over the second one.

4 Consistency and completeness

The subject of this section is an examination of some *a priori* possible definitions of the notions of *sufficient completeness* and *hierarchical consistency* with loose semantics. We start with the most loose semantics: the entire category $Alg(SPEC)$. We will show that the simplest definitions are unacceptable for abstract specification purposes.

All the counter-examples provided in this section are based on the following specification+presentation example. Hopefully, we believe that this counter-example cannot be suspected to be too much unusual, complicated or *ad hoc*.

Example 5 : Let *SPEC* be a specification of natural numbers (for instance the specification given in Example 4) together with a sort $BOOL$ and boolean operations $True$ and $False$. We consider the presentation $PRES$ enriching *SPEC* by an equality predicate *eq*? :

$$
\begin{aligned}
&eq?(0,0) = True\\
&eq?(0, succ(n)) = False\\
&eq?(succ(m), 0) = False\\
&eq?(succ(m), succ(n)) = eq?(m,n)
\end{aligned}
$$

Looking at this presentation $PRES$, we can affirm that a "good notion" of sufficient completeness (resp. hierarchical consistency) should be satisfied by $PRES$. This example is simply written by taking into account each possible value for the arguments of *eq*?, with respect to the constructors of *SPEC*, moreover there are no axioms between constructors (fair presentation [Bid82]).

We may of course imagine more sophisticated presentation examples, in particular examples which add new sorts to *SPEC*. But our goal is simply to prevent the abstract data type researcher from using a naive, rather unrealistic definition of sufficient completeness or hierarchical consistency.

4.1 Sufficient completeness

In the initial approach, sufficient completeness is defined as follows [Gau78].

"*The adjunction morphism associated with the initial algebra is surjective:*"

$$T_SPEC \rightarrow \mathcal{U}(\mathcal{F}(T_SPEC))$$

This condition exactly means that $PRES$ does not add new values to T_SPEC . Remind that $\mathcal{F}(T_SPEC) = T_{SPEC+PRES}$, due to adjunction properties.

fact 6 : The following definition of sufficient completeness is not suitable in the general case:
"$PRES$ is sufficiently complete if and only if for all algebras in $Alg(SPEC)$ the adjunction morphism is surjective".

Using Example 5, we convince ourselves of this fact by considering the *SPEC*-algebra obtained by two copies of $I\!N$. This algebra, ($I\!N \times \{0,1\}$ and $\{True, False\}$) , is not finitely generated, but is an object of $Alg(SPEC)$ by sending the operation-name 0 over the element $(0,0)$, and $succ((n,a)) = (succ(n), a)$. Terms of the form $eq?((n,0),(m,1))$ cannot be evaluated using the $PRES$ axioms of Example 5. Consequently, they add new boolean values, and the adjunction morphism is not surjective.

fact 7 : The following two definitions of sufficient completeness are logically equivalent:

1. the adjunction morphism associated with the initial algebra T_SPEC is surjective
2. for all algebras in $Gen(SPEC)$ the adjunction morphism is surjective.

Proof: $[2 \Longrightarrow 1]$ is trivial because the initial algebra is finitely generated.
$[1 \Longrightarrow 2]$: let A be a finitely generated *SPEC*-algebra. By construction of $\mathcal{F}$, $\mathcal{F}(A)$ is finitely generated over the signature of $SPEC + PRES$. Consequently, the image of the initial morphism via the forgetful functor is surjective:

$$\mathcal{U}(init_A) : \mathcal{U}(\mathcal{F}(T_SPEC)) = \mathcal{U}(T_{SPEC+PRES}) \rightarrow \mathcal{U}(\mathcal{F}(A))$$

Our conclusion results from the commutativity of the following diagram:

$$\begin{array}{ccc} A & \xrightarrow{\text{adjunction}} & U(F(A)) \\ \uparrow \textit{(surjective)} & & \uparrow \textit{(surjective)} \\ T_{SPEC} & \xrightarrow{\textit{(surjective)}} & U(T_{SPEC+PRES}) \end{array}$$

□

Restricting ourselves to finitely generated algebras has several disadvantages. For instance, parameterized presentations require a non finitely generated semantics [ADJ80].

4.2 Hierarchical consistency

In the initial approach, hierarchical consistency is defined as follows:

"*the adjunction morphism associated with the initial algebra is a monomorphism*"

(i.e. is *injective* in our framework).

fact 8 : The following definition of hierarchical consistency is not suitable in the general case:
"$PRES$ is hierarchically consistent if and only if for all algebras in $Alg(SPEC)$ the adjunction morphism is injective".

Let us return to Example 5. If we consider the *SPEC*-algebra $Z\!\!Z$ (which is a non finitely generated algebra), we get the following inconsistency:

$$True = eq?(0,0) = eq?(0, succ(-1)) = False$$

Restricting hierarchical consistency checks to finitely generated algebras does not yield better results:

fact 9 : The following definition of hierarchical consistency is not suitable in the general case:
"$PRES$ is hierarchically consistent if and only if for all algebras in $Gen(SPEC)$ the adjunction morphism is injective".

Using Example 5 again, we consider a finitely generated algebra of the form $\frac{\mathbb{Z}}{n\mathbb{Z}}$, and we get the following inconsistency:

$$True = eq?(0,0) = eq?(0,n) = eq?(0, succ(n-1)) = False$$

These facts prove that "defining sufficient completeness on $Alg(SPEC)$" , "defining hierarchical consistency on $Alg(SPEC)$" or "defining hierarchical consistency on $Gen(SPEC)$" are too strong requirements. Extension from the purely initial semantics to a loose semantics must be done more carefully.

5 Combining presentations

In the remainder of this paper, we simply follow the definitions of sufficient completeness and hierarchical consistency given at the beginning of sections 4.1 and 4.2 (i.e. the initial approach). Given a specification $SPEC$, we consider two presentations $PRES_1$ and $PRES_2$ *with disjoint signatures*.
Let $PRES$ be the union of $PRES_1$ and $PRES_2$, we care about the sufficient completeness and hierarchical consistency of $PRES$. In spite of the strong hypothesis described here, we have sometimes to be careful, as detailed in the following two subsections.

5.1 Sufficient completeness

fact 10 : If $PRES_1$ and $PRES_2$ are both sufficiently complete over $SPEC$, then $PRES = PRES_1 + PRES_2$ remain sufficiently complete. Moreover, under the same hypothesis, $PRES_2$ is sufficiently complete over $SPEC + PRES_1$.

Proof: *(using elementary tools)*
$T_{SPEC+PRES_1+PRES_2}$ is the quotient of $T_{\Sigma+\Sigma_1+\Sigma_2}$ by the smallest congruence containing the close instanciations of the $SPEC + PRES_1 + PRES_2$ axioms [BPW82]. Consequently, it suffices to prove that each $\Sigma + \Sigma_1 + \Sigma_2$-ground-term of sort in S (resp. in $S + S_1$) belongs to the equivalence class of a Σ-term (resp. $\Sigma + \Sigma_1$-term). This can be trivially proved via structural induction. □

Obviously, the converse is false: the sufficient completeness of $PRES$ does not imply the sufficient completeness of $PRES_1$ or $PRES_2$.

5.2 Hierarchical consistency

fact 11 : The hierarchical consistency of $PRES_1$ and $PRES_2$ over $SPEC$ *does not imply* the hierarchical consistency of $PRES = PRES_1 + PRES_2$ over $SPEC$.

Example 12 : Let $SPEC$ be a specification of natural numbers. Let $PRES_1$ be the presentation simply containing the following axiom:

$$succ(n) = 0 \Longrightarrow n = 0$$

$PRES_1$ is clearly consistent (in fact, the premise cannot be satisfied in the initial object, thus this axiom is never applied). Let $PRES_2$ be the presentation adding the operation *pred_* with $[pred(succ(n)) = succ(pred(n)) = n]$. $PRES_2$ is clearly hierarchically consistent over natural numbers (even though it is not sufficiently complete). The union $PRES = PRES_1 + PRES_2$ is not hierarchically consistent because from $succ(pred(0)) = 0$ we get:

$$0 = pred(0) \text{ , which leads to } succ(0) = succ(pred(0)) = 0$$

Another example of the same fact is the following:

Example 13 : Let $PRES_1$ be the presentation described in Example 5 (adding equality predicate to natural numbers), and let $PRES_2$ be the same presentation as Example 12 before (adding *pred*). $PRES_1$ and $PRES_2$ are clearly hierarchically consistent over natural numbers, but the union $PRES = PRES_1 + PRES_2$ is not hierarchically consistent because:

$$True = eq?(0,0) = eq?(0, succ(pred(0))) = False$$

(a similar example was first presented in [EKP80], for abstract implementation purposes).

fact 14 : If $PRES = PRES_1 + PRES_2$ is hierarchically consistent over $SPEC$ then $PRES_1$ and $PRES_2$ are hierarchically consistent over $SPEC$.

Proof: Assume that $PRES_1$ is not consistent: the morphism from T_{SPEC} to $\mathcal{U}(T_{SPEC+PRES_1})$ is not injective. Since the following diagram commutes, the adjunction morphism from T_{SPEC} to $T_{SPEC+PRES_1+PRES_2}$ is not injective:

$$U_1(T_{SPEC+PRES_1}) \longrightarrow \mathcal{U}(T_{SPEC+PRES_1+PRES_2})$$

($PRES_1$ *adjunction morphism*) : $T_{SPEC} \to U_1(T_{SPEC+PRES_1})$

($PRES_1 + PRES_2$ *adjunction morphism*) : $T_{SPEC} \to \mathcal{U}(T_{SPEC+PRES_1+PRES_2})$

$$T_{SPEC}$$

(the horizontal arrow is the forgetful of the adjunction morphism for $PRES_2$ over $SPEC + PRES_1$).
It results that $PRES$ is not hierarchically consistent over $SPEC$. □

fact 15 : If $PRES_1$ and $PRES_2$ are both hierarchically consistent and sufficiently complete over *SPEC*, then $PRES = PRES_1 + PRES_2$ too. Moreover, under the same hypothesis, $PRES_2$ is hierarchically consistent and sufficiently complete over $SPEC + PRES_1$.

(This fact is well known; a demonstration with conditional axioms, including exception handling, can be found in [Ber86]).

6 Loose semantics with "Protect"

Clearly, abstract specifications do not necessarily directly lead to executable specifications. It is often convenient to specify some operations via "universal properties." For instance the subtraction can be specified via:

$$z - y = x \Longleftrightarrow x + y = z$$

Sometimes, such axioms may lead to uncompletely specified presentations, as in the following example.

Example 16 : Let *SPEC* be an initial specification of integers with operations 0, *succ*_, *pred*_, _ + _, _ − _ and _ × _ . Let us specify a presentation *PRES* adding the operation _*div*_ as follows:

$$0 \Longleftarrow (a - (b \times (adivb))) = True$$

$$(a - (b \times (adivb))) < b = True$$

These axioms characterize $(a\,div\,b)$ among all integers *finitely generated* with respect to *succ* and *pred*. However, in the initial model $T_{SPEC+PRES}$, the term $(a\,div\,b)$ is not reached by *succ* and *pred* . Its value is only a unreachable value such that the (unreachable) remainder $(a - (b \times (a\ div\ b)))$ returns the specified boolean values when compared with 0 and b .
Consequently, this presentation is uncompletely specified according to the usual definition of sufficient completeness.

In such examples, the only interesting models are those which do not modify the predefined initial model ($\mathbb{Z}$). This leads to a (loose) semantics where models are those *protecting* predefined sorts [Kam80]. Indeed, when writing relatively large specifications, this semantics seems to be highly suitable (ASL [Wir82] [SW83], PLUSS [Gau84], OBJ [FGJM85], LARCH [GH83]...).

Let us define the associated category:

Definition 17 : (The "Protect" category)
Let *SPEC* be a specification and let *PRES* be a presentation over *SPEC*. The category of *PRES*-models protecting *SPEC* is the full subcategory of $Alg(SPEC + PRES)$ whose objects are the $SPEC + PRES$-algebras A such that $\mathcal{U}(A)$ is isomorphic to the initial predefined algebra T_{SPEC} . We denote this category by $Prot(SPEC, PRES)$.

Notice that the object class of $Prot(SPEC, PRES)$ can be empty.

fact 18 : If $Prot(SPEC, PRES)$ is not an empty category, then $PRES$ is hierarchically consistent over $SPEC$.

(Here, consistency is defined with respect to the initial algebra T_{SPEC} only)

Proof: If $T_{SPEC+PRES}$ is inconsistent over T_{SPEC}, then a fortiori all $SPEC + PRES$-algebras are inconsistent over T_{SPEC} (because $T_{SPEC+PRES}$ is minimal). □

fact 19 : Even if $PRES$ is consistent over $SPEC$, $Prot(SPEC, PRES)$ may be empty.

Example 20 : Let $SPEC$ be the boolean specification with $True$ and $False$. Let $PRES$ be a specification of $SET(BOOL)$ with $\emptyset$, *insert*, $\in$ and *choose* :

$True \in \emptyset = False$
$False \in \emptyset = False$
$b \in insert(b', X) = (b{=}b') \text{ or } b \in X$
$choose(X) \in X = True$

This specification is clearly hierarchically consistent (even though it is not sufficiently complete). However, the Protect category is empty, because the term $choose(\emptyset)$ can neither be equal to $True$ nor to $False$ (both choices induce $True = False$).
(Fortunately, this example can be easily specified without inconsistency using abstract data types with exception handling [Bid84] [GDLE84] [BBC86b] [Ber86], or with partial functions [BW82].)

fact 21 : Even if $Prot(SPEC, PRES)$ contains models, it has not necessarily an initial object.

Example 22 : Let $SPEC$ be the boolean specification with $True$ and $False$. Let $PRES$ be the presentation adding the constant operation *maybe*, without any axiom. $Prot(SPEC, PRES)$ contains two models, no one is initial.

fact 23 : If $PRES$ is sufficiently complete over $SPEC$, then either the category $Prot(SPEC, PRES)$ has an initial object, either it is empty.

Proof: If $PRES$ is consistent, then the initial model $T_{SPEC+PRES}$ belongs to $Prot(SPEC, PRES)$; it is then necessarily initial in $Prot(SPEC, PRES)$. If $PRES$ is not hierarchically consistent, then Fact 18 implies that $Prot(SPEC, PRES)$ has no object. □

fact 24 : There are presentations $PRES$ which are not sufficiently complete over $SPEC$, such that $Prot(SPEC, PRES)$ is not empty and has an initial object.

It suffices to refer to Example 16, where the axioms characterize *div* by a "universal property among integers." The division is incompletely specified according to classical initial definition of sufficient completeness, but $Prot(SPEC, PRES)$ only contains one model ($\mathbb{Z}$) which is necessarily initial.

7 Conclusion

We have investigated how a hierarchical approach of abstract data types, with the notions of *hierarchical consistency* and *sufficient completeness*, could be defined when dealing with so-called *loose semantics*. The results shown in sections 2 to 5 seem to be somewhat pessimistic:

- The synthesis functor is "implementation dependent" with respect to the predefined specification (Fact 3).
- Sufficient completeness cannot be checked on all models (Fact 6).
- Hierarchical consistency cannot be checked on all models (Fact 8).
- Hierarchical consistency cannot be checked on all finitely generated models, a smaller class of models must be investigated (Fact 9).
- Combining hierarchically consistent presentations does not result on a hierarchically consistent presentation (Fact 11).

However, we showed some positive results:

- Checking sufficient completeness on all finitely generated algebras is equivalent to check it on the initial algebra only (Fact 7).
- Combining sufficiently complete presentations results on sufficiently complete presentations (Fact 10); the same occurs for presentations that are both sufficiently complete and hierarchically consistent (Fact 15).

In the last section (Section 6), we defined the category of models *protecting predefined sorts*. We have investigated the relations between the classical notions of completeness/consistency and the elementary properties of this category:

- The category is empty if the presentation is not hierarchically consistent, but the converse is false (Facts 18 and 19).
- The category has not necessarily initial models (Fact 21).
  It has initial models if the presentation is sufficiently complete and hierarchically consistent, but the converse is false (Facts 23 and 24).

In conclusion: From facts 3, 6, 8, 9 and 11, we showed that the synthesis functor of classical abstract data types "does not always preserve philosophy" when dealing with loose semantics. Moreover, with a loose semantics based on protection of predefined sorts, the corresponding category has few systematic relations with sufficient completeness or hierarchical consistency (facts 18 to 24).

Acknowledgements: It is a pleasure to express gratitude to Michel Bidoit, Christine Choppy and Marie-Claude Gaudel for encouragements to write this paper and careful proof readings. The title was suggested by Stephane Kaplan.

References

[ADJ76] Goguen J., Thatcher J., Wagner E. : "*An initial algebra approach to the specification, correctness, and implementation of abstract data types*", Current Trends in Programming Methodology, Vol.4, Yeh Ed. Prentice Hall, 1978. Also : IBM Report RC 6487, Oct. 1976.

[ADJ80] Ehrig H., Kreowski H., Thatcher J., Wagner J., Wright J. : "*Parameterized data types in algebraic specification langages*", Proc. 7th ICALP, July 1980.

[BBC86a] Bernot G., Bidoit M., Choppy C. : "*Abstract implementations and correctness proofs*", Proc. 3rd STACS, January 1986, Springer-Verlag LNCS 210, January 1986. Also : LRI Report 250, Orsay, Dec. 1985.

[BBC86b] Bernot G., Bidoit M., Choppy C. : "*Abstract data types with exception handling : an initial approach based on a distinction between exceptions and errors*", Theoretical Computer Science, Vol.46, No.1, p.13-45, November 1986.

[Ber86] Bernot G. : "*Une sémantique algébrique pour une spécification différenciée des exceptions et des erreurs : application à l'implémentation et aux primitives de structuration des spécifications formelles*", Thèse de troisième cycle, LRI, Université de Paris-Sud, Orsay, Février 1986.

[Bid82] Bidoit M. : "*Algebraic data types: structured specifications and fair presentations*", Proc. AFCET Symposium on Mathematics for Computer Science, Paris, March 1982.

[Bid84] Bidoit M. : "*Algebraic specification of exception handling by means of declarations and equations*", Proc. 11th ICALP, Springer-Verlag LNCS 172, July 1984.

[BPW82] Broy M., Pair C., Wirsing M. : "*A systematic study of models of abstract data types*", Theoretical Computer Sciences, p. 139-174, vol. 33, October 1984.

[BW82] Broy M., Wirsing M. : "*Partial abstract data types*", Acta Informatica, Vol.18-1, Nov. 1982.

[EKMP80] Ehrig H., Kreowski H., Mahr B., Padawitz P. : "*Algebraic implementation of abstract data types*", Theoretical Computer Science, Oct. 1980.

[EKP80] Ehrig H., Kreowski H., Padawitz P. : "*Algebraic implementation of abstract data types: concept, syntax, semantics and correctness*", Proc. ICALP, Springer-Verlag LNCS 85, 1980.

[FGJM85] Futatsugi K., Goguen J., Jouannaud J-P., Meseguer J. : "*Principles of OBJ2*", Proc. 12th ACM Symp. on Principle of Programming Languages, New Orleans, January 1985.

[Gau78] Gaudel M-C. : "*Spécifications incomplètes mais suffisantes de la représentation des types abstraits*", Laboria Report 320, 1978.

[Gau84] Gaudel M-C. : "*A first introduction to PLUSS*", LRI Report, Orsay, December 1984.

[GDLE84] Gogolla M., Drosten K., Lipeck U., Ehrich H.D. : "*Algebraic and operational semantics of specifications allowing exceptions and errors*", Theoretical Computer Science 34, North Holland, 1984.

[GH83] Guttag J.V., Horning J.J. : "*An introduction to the LARCH shared language*", Proc. IFIP 83, REA Mason ed., North Holland Publishing Company, 1983.

[Kam80] Kamin S. : "*Final data type specifications : a new data type specification method*", Proc. of the 7th POPL Conference, 1980.

[SW82] Sannella D., Wirsing M. : "*Implementation of parameterized specifications*", Report CSR-103-82, Department of Computer Science, University of Edinburgh.

[SW83] Sannella D., Wirsing M. : "*A kernel language for algebraic specification and implementation*", Proc. Intl. Conf. on Foundations of computation Theory, Springer-Verlag, LNCS 158, 1983.

[Wir82] Wirsing M. : "*Structured algebraic specifications*", Proc. of AFCET Symposium on Mathematics for Computer Science, Paris, March 1982.